# Chapter 7
# Arrays and ArrayLists

Java How to Program, 11/e, Global Edition

Questions? E-mail paul.deitel@deitel.com

# OBJECTIVES

In this chapter you'll:

- Learn what arrays are.
- Use arrays to store data in and retrieve data from lists and tables of values.
- Declare arrays, initialize arrays and refer to individual elements of arrays.
- Iterate through arrays with the enhanced `for` statement.
- Pass arrays to methods.

# OBJECTIVES

- Declare and manipulate multidimensional arrays.
- Use variable-length argument lists.
- Read command-line arguments into a program.
- Build an object-oriented instructor gradebook class.
- Perform common array manipulations with the methods of class `Arrays`.
- Use class `ArrayList` to manipulate a dynamically resizable array-like data structure.

# OUTLINE

# OUTLINE (cont.)

# OUTLINE (cont.)

# OUTLINE (cont.)

# 7.1 Introduction

- Data structures
  - Collections of related data items.
  - Discussed in depth in Chapters 16–21.
- Array objects
  - Data structures consisting of related data items of the same type.
  - Make it convenient to process related groups of values.
  - Remain the same length once they are created.
- Enhanced `for` statement for iterating over an array or collection of data items.
- Variable-length argument lists
  - Can create methods are with varying numbers of arguments.
- Process command-line arguments in method `main`.

# 7.1 Introduction (Cont.)

- Common array manipulations with `static` methods of class `Arrays` from the `java.util` package.
- `ArrayList` collection
  - Similar to arrays
  - Dynamic resizing
    - resize as necessary to accommodate more or fewer elements

# 7.1 Introduction (Cont.)

- Java SE 8
  - After reading Chapter 17, Java SE 8 Lambdas and Streams, you'll be able to reimplement many of Chapter 7's examples in a more concise and elegant manner, and in a way that makes them easier to parallelize to improve performance on today's multi-core systems.
  - Recall from the Preface that Chapter 17 is keyed to many earlier sections of the book so that you can conveniently cover lambdas and streams if you'd like. If so, we recommend that after you complete Chapter 7, you read Sections 17.1–17.7, which introduce the lambdas and streams concepts and use them to rework examples from Chapters 4–7.

# 7.2 Arrays

- Array
  - Group of variables (called elements) containing values of the same type.
  - Arrays are objects so they are reference types.
  - Elements can be either primitive or reference types.
- Refer to a particular element in an array
  - Use the element's index.
  - Array-access expression—the name of the array followed by the index of the particular element in square brackets, [ ].
- The first element in every array has index zero.
- The highest index in an array is one less than the number of elements in the array.
- Array names follow the same conventions as other variable names.

# 7.2 Arrays (Cont.)

- An index must be a nonnegative integer.
  - Can use an expression as an index.
- An indexed array name is an array-access expression.
  - Can be used on the left side of an assignment to place a new value into an array element.
- Every array object knows its own length and stores it in a `length` instance variable.
  - `length` cannot be changed because it's a `final` variable.

## Common Programming Error 7.1

An index must be an `int` value or a value of a type that can be promoted to `int`—namely, `byte`, `short` or `char`, but not `long`; otherwise, a compilation error occurs.

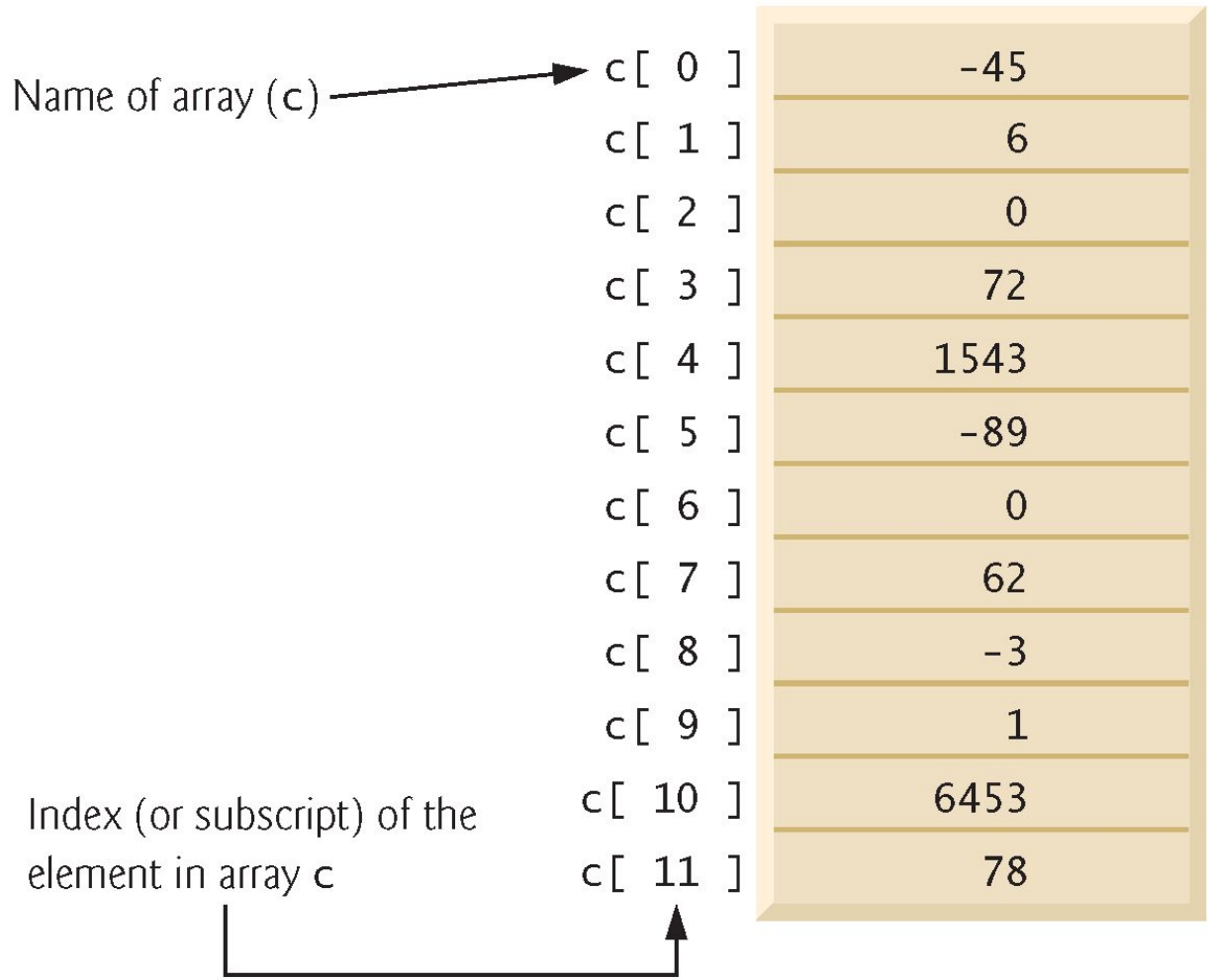


**Fig. 7.1** | A 12-element array.

# 7.3 Declaring and Creating Arrays

- Array objects
  - Created with keyword `new`.
  - You specify the element type and the number of elements in an array-creation expression, which returns a reference that can be stored in an array variable.
- Declaration and array-creation expression for an array of 12 `int` elements

```
int[] c = new int[12];
```

- Can be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[12]; // creates the array
```

# 7.3 Declaring and Creating Arrays (Cont.)

- In a declaration, *square brackets* following a type indicate that a variable will refer to an array (i.e., store an array *reference*).
- When an array is created, each element of the array receives a default value
  - Zero for the numeric primitive-type elements, `false` for `boolean` elements and `null` for references.

## Common Programming Error 7.2

In an array declaration, specifying the number of elements in the square brackets of the declaration (e.g., `int[12] c;`) is a syntax error.

# 7.3 Declaring and Creating Arrays (Cont.)

- When the element type and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables.
  - For readability, declare only one variable per declaration.

## Good Programming Practice 7.1

For readability, declare only one variable per declaration. Keep each declaration on a separate line, and include a comment describing the variable being declared.

## Common Programming Error 7.3

Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration `int[] a, b, c;`. If `a`, `b` and `c` should be declared as array variables, then this declaration is correct—placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only `a` is intended to be an array variable, and `b` and `c` are intended to be individual `int` variables, then this declaration is incorrect—the declaration `int a[], b, c;` would achieve the desired result.

# 7.3 Declaring and Creating Arrays (Cont.)

- Every element of a primitive-type array contains a value of the array's declared element type.
  - Every element of an `int` array is an `int` value.
- Every element of a reference-type array is a reference to an object of the array's declared element type.
  - Every element of a `String` array is a reference to a `String` object.

# 7.4 Examples Using Arrays

- This section presents several examples that demonstrate declaring arrays, creating arrays, initializing arrays and manipulating array elements.

# 7.4.1 Creating and Initializing an Array

- Fig. 7.2 uses keyword `new` to create an array of 10 `int` elements, which are initially zero (the default initial value for `int` variables).

```
1  // Fig. 7.2: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // declare variable array and initialize it with an array object
7        int[] array = new int[10]; // create the array object
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; counter++) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

**Fig. 7.2** | Initializing the elements of an array to default values of zero. (Part 1 of 2.)

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

**Fig. 7.2** | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

# 7.4.2 Using an Array Initializer

- Array initializer
  - A comma-separated list of expressions (called an initializer list) enclosed in braces.
  - Used to create an array and initialize its elements.
  - Array length is determined by the number of elements in the initializer list.

```
int[] n = {10, 20, 30, 40, 50};
```

    - Creates a five-element array with index values `0`–`4`.
- Compiler counts the number of initializers in the list to determine the size of the array
  - Sets up the appropriate `new` operation "behind the scenes."

```
1  // Fig. 7.3: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // initializer list specifies the initial value for each element
7        int[] array = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
8
9        System.out.printf("%s%8s%n", "Index", "Value"); // column headings
10
11       // output each array element's value
12       for (int counter = 0; counter < array.length; counter++) {
13          System.out.printf("%5d%8d%n", counter, array[counter]);
14       }
15    }
16 }
```

**Fig. 7.3** | Initializing the elements of an array with an array initializer. (Part 1 of 2.)

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

**Fig. 7.3** | Initializing the elements of an array with an array initializer. (Part 2 of 2.)

# 7.4.3 Calculating the Values to Store in an Array

- The application in Fig. 7.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20).

```
1  // Fig. 7.4: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        final int ARRAY_LENGTH = 10; // declare constant
7        int[] array = new int[ARRAY_LENGTH]; // create array
8
9        // calculate value for each array element
10       for (int counter = 0; counter < array.length; counter++) {
11          array[counter] = 2 + 2 * counter;
12       }
13
14       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
15
16       // output each array element's value
17       for (int counter = 0; counter < array.length; counter++) {
18          System.out.printf("%5d%8d%n", counter, array[counter]);
19       }
20    }
21 }
```

**Fig. 7.4** | Calculating the values to be placed into the elements of an array. (Part 1 of 2.)

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

**Fig. 7.4** | Calculating the values to be placed into the elements of an array. (Part 2 of 2.)

# 7.4 Examples Using Arrays (Cont.)

- `final` variables must be initialized before they are used and cannot be modified thereafter.
- An attempt to modify a `final` variable after it's initialized causes a compilation error
  - `cannot assign a value to final variable` *variableName*
- An attempt to access the value of a `final` variable before it's initialized causes a compilation error
  - `variable` *variableName* `might not have been initialized`

## Good Programming Practice 7.2

Constant variables also are called **named constants**. They often make programs more readable—a named constant such as ARRAY_LENGTH clearly indicates its purpose, whereas a literal value such as 10 could have different meanings based on its context.

## Good Programming Practice 7.3

Constants use all uppercase letters by convention and multiword named constants should have each word separated from the next with an underscore (_) as in ARRAY_LENGTH.

## Common Programming Error 7.4

Assigning a value to a previously initialized `final` variable is a compilation error. Similarly, attempting to access the value of a `final` variable before it's initialized results in a compilation error like, "`variable` variableName `might not have been initialized`."

# 7.4.4 Summing the Elements of an Array

- Figure 7.5 sums the values contained in a 10-element integer array.
- Often, the elements of an array represent a series of values to be used in a calculation.

```
1  // Fig. 7.5: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray {
5     public static void main(String[] args) {
6        int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int counter = 0; counter < array.length; counter++) {
11          total += array[counter];
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

```
Total of array elements: 849
```

**Fig. 7.5** | Computing the sum of the elements of an array.

# 7.4.5 Using Bar Charts to Display Array Data Graphically

- Many programs present data to users in a graphical manner.
- Numeric values are often displayed as bars in a bar chart.
  - Longer bars represent proportionally larger numeric values.
- A simple way to display numeric data is with a bar chart that shows each numeric value as a bar of asterisks (*).
- Format specifier `%02d` indicates that an `int` value should be formatted as a field of two digits.
  - The `0` flag displays a leading `0` for values with fewer digits than the field width (2).

```
1  // Fig. 7.6: BarChart.java
2  // Bar chart printing program.
3
4  public class BarChart {
5     public static void main(String[] args) {
6        int[] array = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
7
8        System.out.println("Grade distribution:");
9
10       // for each array element, output a bar of the chart
11       for (int counter = 0; counter < array.length; counter++) {
12          // output bar label ("00-09: ", ..., "90-99: ", "100: ")
13          if (counter == 10) {
14             System.out.printf("%5d: ", 100);
15          }
16          else {
17             System.out.printf("%02d-%02d: ",
18                counter * 10, counter * 10 + 9);
19          }
```

**Fig. 7.6** | Bar chart printing program. (Part 1 of 2.)

```
20
21             // print bar of asterisks
22             for (int stars = 0; stars < array[counter]; stars++) {
23                System.out.print("*");
24             }
25
26             System.out.println();
27          }
28       }
29    }
```

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

**Fig. 7.6** | Bar chart printing program. (Part 2 of 2.)

# 7.4.6 Using the Elements of an Array as Counters

- Sometimes, programs use counter variables to summarize data, such as the results of a survey.
- Fig. 6.7 used separate counters in a die-rolling program to track the number of occurrences of each side of a six-sided die as the program rolled the die 60,000,000 times.
- Fig. 7.7 shows an array version of this application.
- Array `frequency` must be large enough to store six counters.
  - We use a seven-element array in which we ignore `frequency[0]`
  - More logical to have the face value 1 increment `frequency[1]` than `frequency[0]`.

```
1  // Fig. 7.7: RollDie.java
2  // Die-rolling program using arrays instead of switch.
3  import java.security.SecureRandom;
4
5  public class RollDie {
6     public static void main(String[] args) {
7        SecureRandom randomNumbers = new SecureRandom();
8        int[] frequency = new int[7]; // array of frequency counters
9
10       // roll die 60,000,000 times; use die value as frequency index
11       for (int roll = 1; roll <= 60_000_000; roll++) {
12          ++frequency[1 + randomNumbers.nextInt(6)];
13       }
14
15       System.out.printf("%s%10s%n", "Face", "Frequency");
16
17       // output each array element's value
18       for (int face = 1; face < frequency.length; face++) {
19          System.out.printf("%4d%10d%n", face, frequency[face]);
20       }
21    }
22 }
```

**Fig. 7.7** | Die-rolling program using arrays instead of `switch`. (Part 1 of 2.)

```
Face Frequency
   1   9995532
   2  10003079
   3  10000564
   4  10000726
   5   9998994
   6  10001105
```

**Fig. 7.7** | Die-rolling program using arrays instead of `switch`. (Part 2 of 2.)

# 7.4.7 Using Arrays to Analyze Survey Results

- Figure 7.8 uses arrays to summarize the results of data collected in a survey:
  - *Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.*
- Array `responses` is a 20-element `int` array of the survey responses.
- 6-element array `frequency` counts the number of occurrences of each response (1 to 5).
  - Each element is initialized to zero by default.
  - We ignore `frequency[0]`.

```
1  // Fig. 7.8: StudentPoll.java
2  // Poll analysis program.
3
4  public class StudentPoll {
5     public static void main(String[] args) {
6        // student response array (more typically, input at runtime)
7        int[] responses =
8           {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 14};
9        int[] frequency = new int[6]; // array of frequency counters
10
```

**Fig. 7.8** | Poll analysis program. (Part 1 of 3.)

```
11          // for each answer, select responses element and use that value
12          // as frequency index to determine element to increment
13          for (int answer = 0; answer < responses.length; answer++) {
14             try {
15                ++frequency[responses[answer]];
16             }
17             catch (ArrayIndexOutOfBoundsException e) {
18                System.out.println(e); // invokes toString method
19                System.out.printf("   responses[%d] = %d%n%n",
20                   answer, responses[answer]);
21             }
22          }
```

**Fig. 7.8** | Poll analysis program. (Part 2 of 3.)

```
23
24          System.out.printf("%s%10s%n", "Rating", "Frequency");
25
26          // output each array element's value
27          for (int rating = 1; rating < frequency.length; rating++) {
28             System.out.printf("%6d%10d%n", rating, frequency[rating]);
29          }
30       }
31    }
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

**Fig. 7.8** | Poll analysis program. (Part 3 of 3.)

# 7.4.7 Using Arrays to Analyze Survey Results (Cont.)

- If a piece of data in the `responses` array is an invalid value, such as 14, the program attempts to add 1 to `frequency[14]`, which is outside the bounds of the array.
  - Java doesn't allow this.
  - JVM checks array indices to ensure that they are greater than or equal to 0 and less than the length of the array—this is called bounds checking.
  - If a program uses an invalid index, Java generates a so-called exception to indicate that an error occurred in the program at execution time.

# 7.5 Exception Handling: Processing the Incorrect Response

- An exception indicates a problem that occurs while a program executes.
- The name “exception” suggests that the problem occurs infrequently—if the “rule” is that a statement normally executes correctly, then the problem represents the “exception to the rule.”
- Exception handling helps you create fault-tolerant programs that can resolve (or handle) exceptions.

# 7.5 Exception Handling: Processing the Incorrect Response (Cont.)

- When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.

# 7.5.1 The `try` Statement

- To handle an exception, place any code that might throw an exception in a `try` statement.
- The `try` block contains the code that might throw an exception.
- The `catch` block contains the code that *handles* the exception if one occurs. You can have many `catch` blocks to handle different *types* of exceptions that might be thrown in the corresponding `try` block.

# 7.5.2 Executing the `catch` Block

- When the program encounters the invalid value 14 in the responses array, it attempts to add 1 to `frequency[14]`, which is *outside* the bounds of the array—the `frequency` array has only six elements (with indexes 0–5).
- Because array bounds checking is performed at execution time, the JVM generates an *exception*—specifically an `ArrayIndexOutOfBoundsException` to notify the program of this problem.
- At this point the `try` block terminates and the `catch` block begins executing—if you declared any local variables in the `try` block, they're now out of scope.

# 7.5.2 Executing the `catch` Block (Cont.)

- The `catch` block declares an exception parameter (e) of type (`IndexOutOfRangeException`).
- Inside the `catch` block, you can use the parameter's identifier to interact with a caught exception object.

## Error-Prevention Tip 7.1

When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This will prevent `ArrayIndexOutOfBoundsExceptions` if your program is correct.

## Software Engineering Observation 7.1

Systems in industry that have undergone extensive testing are still likely to contain bugs. Our preference for industrial-strength systems is to catch and deal with runtime exceptions, such as `ArrayIndexOutOfBoundsExceptions`, to ensure that a system either stays up and running or degrades gracefully, and to inform the system's developers of the problem.

# 7.5.3 `toString` Method of the Exception Parameter

- The exception object's `toString` method returns the error message that's implicitly stored in the exception object.
- The exception is considered handled when program control reaches the closing right brace of the `catch` block.

# 7.6 Case Study: Card Shuffling and Dealing Simulation

- Examples thus far used arrays containing 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. 7.9) 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. 7.9: 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. 7.9** | Card class represents a playing card.

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

- Class `DeckOfCards` (Fig. 7.10) 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 `Cards` in the deck.
  - Loops through all 52 `Cards` (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. 7.10: 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. 7.10** | 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. 7.10** | `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. 7.10** | `DeckOfCards` class represents a deck of playing cards. (Part 3 of 4.)

```
43
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. 7.10** | `DeckOfCards` class represents a deck of playing cards. (Part 4 of 4.)

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

- Figure 7.11 demonstrates class `DeckOfCards`.
- When a `Card` is output as a `String`, the `Card`'s `toString` method is implicitly invoked.

```
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 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. 7.11** | 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. 7.11** | Card shuffling and dealing. (Part 2 of 2.)

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

### *Preventing `NullPointerExceptions`*

- In Fig. 7.10, 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 Enhanced for Statement

- Enhanced for statement
  - Iterates through the elements of an array without using a counter.
  - Avoids the possibility of "stepping outside" the array.
  - Also works with the Java API's prebuilt collections (see Section 7.14).
- Syntax:

```
for (parameter : arrayName) {
   statement
}
```

  where *parameter* has a type and an identifier and *arrayName* is the array through which to iterate.
- Parameter type must be consistent with the array's element type.
- The enhanced for statement simplifies the code for iterating through an array.

```
1  // Fig. 7.12: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest {
5     public static void main(String[] args) {
6        int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7        int total = 0;
8
9        // add each element's value to total
10       for (int number : array) {
11          total += number;
12       }
13
14       System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

```
Total of array elements: 849
```

**Fig. 7.12** | Using the enhanced `for` statement to total integers in an array.

# 7.7 Enhanced `for` Statement (Cont.)

- The enhanced `for` statement can be used *only* to obtain array elements
  - It *cannot* be used to *modify* elements.
  - To modify elements, use the traditional counter-controlled `for` statement.
- Can be used in place of the counter-controlled `for` statement if you don't need to access the index of the element.

## Error-Prevention Tip 7.2

The enhanced `for` statement simplifies iterating through an array. This makes the code more readable and eliminates several error possibilities, such as improperly specifying the control variable's initial value, the loop-continuation test and the increment expression.

# 7.7 Enhanced `for` Statement (Cont.)

***Java SE 8***

- The `for` statement and the enhanced for statement each iterate sequentially from a starting value to an ending value.
- In Chapter 17, Java SE 8 Lambdas and Streams, you'll learn about class `Stream` and its `forEach` method.
- Working together, these provide an elegant, more concise and less error prone means for iterating through collections so that some of the iterations may occur in parallel with others to achieve better multi-core system performance.

# 7.8 Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets.
  - Since every array object "knows" its own length, we need not pass the array length as an additional argument.
- To receive an array, the method's parameter list must specify an *array parameter*.
- When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference.
- When an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's value.
  - Such primitive values are called scalars or scalar quantities.

```
1  // Fig. 7.13: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray {
5     // main creates array and calls modifyArray and modifyElement
6     public static void main(String[] args) {
7        int[] array = {1, 2, 3, 4, 5};
8
9        System.out.printf(
10          "Effects of passing reference to entire array:%n" +
11          "The values of the original array are:%n");
12
13       // output original array elements
14       for (int value : array) {
15          System.out.printf("   %d", value);
16       }
17
```

**Fig. 7.13** | Passing arrays and individual array elements to methods. (Part 1 of 4.)

```
18          modifyArray(array); // pass array reference
19          System.out.printf("%n%nThe values of the modified array are:%n");
20
21          // output modified array elements
22          for (int value : array) {
23             System.out.printf("   %d", value);
24          }
25
26          System.out.printf(
27             "%n%nEffects of passing array element value:%n" +
28             "array[3] before modifyElement: %d%n", array[3]);
29
30          modifyElement(array[3]); // attempt to modify array[3]
31          System.out.printf(
32             "array[3] after modifyElement: %d%n", array[3]);
33       }
34
```

**Fig. 7.13** | Passing arrays and individual array elements to methods. (Part 2 of 4.)

```
35      // multiply each element of an array by 2
36      public static void modifyArray(int[] array2) {
37         for (int counter = 0; counter < array2.length; counter++) {
38            array2[counter] *= 2;
39         }
40      }
41
42      // multiply argument by 2
43      public static void modifyElement(int element) {
44         element *= 2;
45         System.out.printf(
46            "Value of element in modifyElement: %d%n", element);
47      }
48   }
```

**Fig. 7.13** | Passing arrays and individual array elements to methods. (Part 3 of 4.)

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

**Fig. 7.13** | Passing arrays and individual array elements to methods. (Part 4 of 4.)

# 7.9 Pass-By-Value vs. Pass-By-Reference

- Pass-by-value (sometimes called call-by-value)
  - A copy of the argument's *value is passed to the called method.*
  - The called method works exclusively with the copy.
  - Changes to the called method's copy do not affect the original variable's value in the caller.
- Pass-by-reference (sometimes called call-by-reference)
  - The called method can access the argument's value in the caller directly and modify that data, if necessary.
  - Improves performance by eliminating the need to copy possibly large amounts of data.

# 7.9 Pass-By-Value vs. Pass-By-Reference (Cont.)

- All arguments in Java are passed by value.
- A method call can pass two types of values to a method
  - Copies of primitive values
  - Copies of references to objects
- Objects cannot be passed to methods.
- If a method modifies a reference-type parameter so that it refers to another object, only the parameter refers to the new object
  - The reference stored in the caller's variable still refers to the original object.
- Although an object's reference is passed by value, a method can still interact with the referenced object by calling its `public` methods using the copy of the object's reference.
  - The parameter in the called method and the argument in the calling method refer to the same object in memory.

## Performance Tip 7.1

Passing references to arrays, instead of the array objects themselves, makes sense for performance reasons. Because Java arguments are passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the copies of the elements.

# 7.10 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 7.12, 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    }
```

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

```
18
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      }
```

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

```
113
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.)

# 7.10 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 7.2

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

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

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

```
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.)

# 7.10 Case Study: Class GradeBook Using an Array to Store Grades (Cont.)

***Java SE 8***

- In Chapter 17, Java SE 8 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 iteration statements.
- In Chapter 23, Concurrency, the example of Fig. 23.29 uses `stream` method `summaryStatistics` to perform all of these operations in one method call.

# 7.11 Multidimensional Arrays

- Two-dimensional arrays are often used to represent tables of values with data arranged in *rows* and *columns*.
- Identify each table element with two indices.
  - By convention, the first identifies the element's row and the second its column.
- Multidimensional arrays can have more than two dimensions.
- Java does not support multidimensional arrays directly
  - Allows you to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect.
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array.

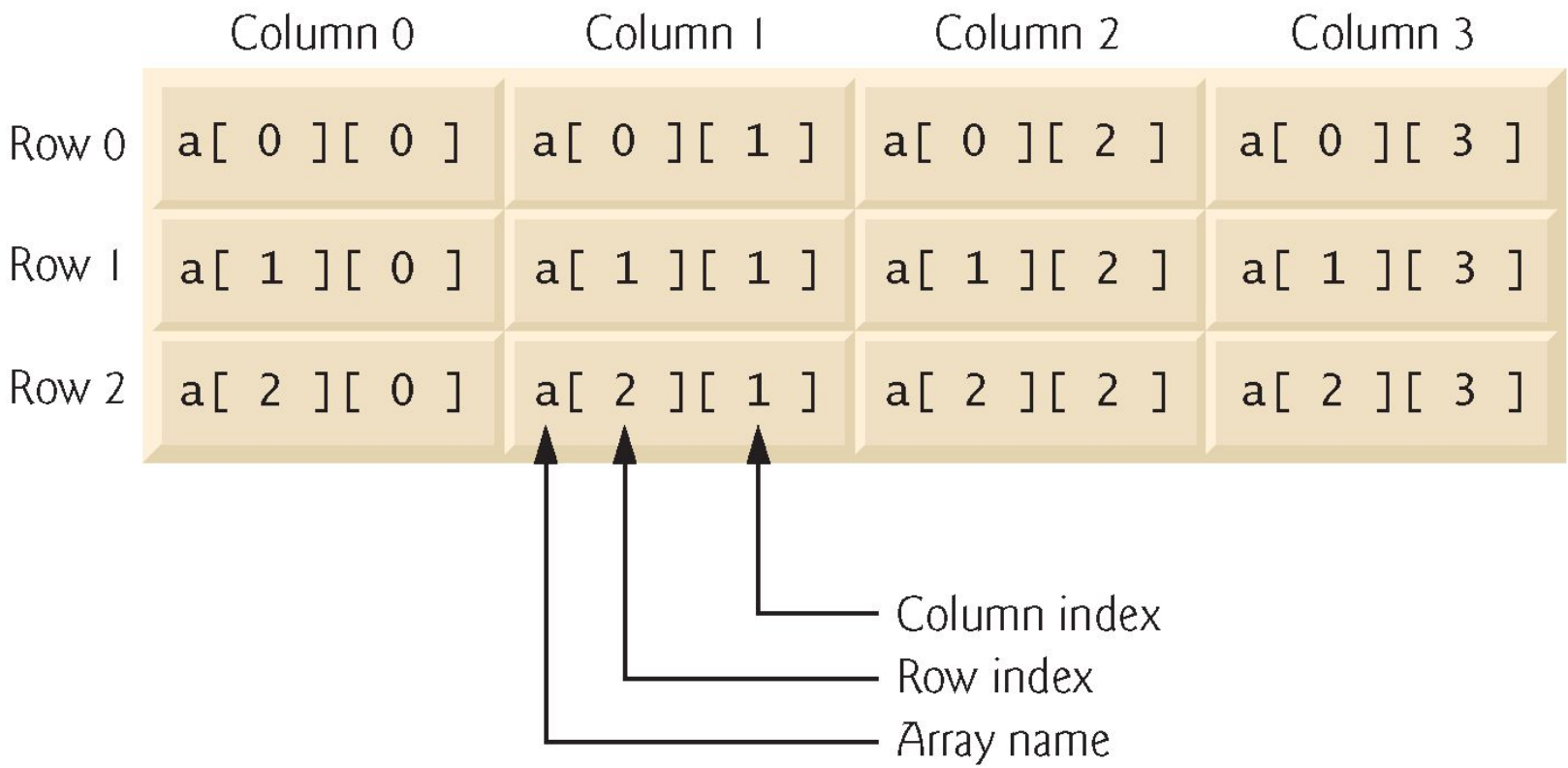


**Fig. 7.16** | Two-dimensional array with three rows and four columns.

# 7.11 Multidimensional Arrays (Cont.)

- Multidimensional arrays can be initialized with array initializers in declarations.
- A two-dimensional array b with two rows and two columns could be declared and initialized with nested array initializers as follows:

```
int[][] b = {{1, 2}, {3, 4}};
```

  - The initial values are *grouped by row* in braces.
  - The number of nested array initializers (represented by sets of braces within the outer braces) determines the number of *rows*.
  - The number of initializer values in the nested array initializer for a row determines the number of *columns* in that row.
  - *Rows can have different lengths.*

# 7.11 Multidimensional Arrays (Cont.)

- The lengths of the rows in a two-dimensional array are not required to be the same:

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

  - Each element of `b` is a reference to a one-dimensional array of `int` variables.
  - The `int` array for row `0` is a one-dimensional array with two elements (`1` and `2`).
  - The `int` array for row `1` is a one-dimensional array with three elements (`3`, `4` and `5`).

# 7.11 Multidimensional Arrays (Cont.)

- A multidimensional array with the same number of columns in every row can be created with an array-creation expression.

```
int[][] b = new int[3][4];
```

  - 3 rows and 4 columns.
- The elements of a multidimensional array are initialized when the array object is created.
- A multidimensional array in which each row has a different number of columns can be created as follows:

```
int[][] b = new int[2][];   // create 2 rows
b[0] = new int[5]; // create 5 columns for row 0
b[1] = new int[3]; // create 3 columns for row 1
```

  - Creates a two-dimensional array with two rows.
  - Row 0 has five columns, and row 1 has three columns.

# 7.11 Multidimensional Arrays (Cont.)

- Figure 7.17 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to traverse the arrays.

```
1  // Fig. 7.17: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray {
5     // create and output two-dimensional arrays
6     public static void main(String[] args) {
7        int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
8        int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
9
10       System.out.println("Values in array1 by row are");
11       outputArray(array1); // displays array1 by row
12
13       System.out.printf("%nValues in array2 by row are%n");
14       outputArray(array2); // displays array2 by row
15    }
```

**Fig. 7.17** | Initializing two-dimensional arrays. (Part 1 of 3.)

```
16
17      // output rows and columns of a two-dimensional array
18      public static void outputArray(int[][] array) {
19         // loop through array's rows
20         for (int row = 0; row < array.length; row++) {
21            // loop through columns of current row
22            for (int column = 0; column < array[row].length; column++) {
23               System.out.printf("%d  ", array[row][column]);
24            }
25
26            System.out.println();
27         }
28      }
29   }
```

**Fig. 7.17** | Initializing two-dimensional arrays. (Part 2 of 3.)

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

**Fig. 7.17** | Initializing two-dimensional arrays. (Part 3 of 3.)

# 7.12 Case Study: Class GradeBook Using a Two-Dimensional Array

- In most semesters, students take several exams.
- Figure 7.18 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.
- In this example, we use a ten-by-three array containing ten students' grades on three exams.

```
1  // Fig. 7.18: 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    }
```

**Fig. 7.18** | GradeBook class using a two-dimensional array to store grades. (Part 1 of 9.)

```
18
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 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      }
```

**Fig. 7.18** | GradeBook class using a two-dimensional array to store grades. (Part 2 of 9.)

```
37
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      }
```

**Fig. 7.18** | GradeBook class using a two-dimensional array to store grades. (Part 3 of 9.)

```
56
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      }
```

**Fig. 7.18** | GradeBook class using a two-dimensional array to store grades. (Part 4 of 9.)

```
75
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. 7.18** | 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. 7.18** | 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. 7.18** | 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. 7.18** | 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. 7.18** | GradeBook class using a two-dimensional array to store grades. (Part 9 of 9.)

```
1  // Fig. 7.19: 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. 7.19** | 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. 7.19** | `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. 7.19** | `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. 7.19** | `GradeBookTest` creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 4 of 4.)

# 7.13 Variable-Length Argument Lists

- Variable-length argument lists
  - Can be used to create methods that receive an unspecified number of arguments.
  - Parameter type followed by an ellipsis (`...`) indicates that the method receives a variable number of arguments of that particular type.
  - The ellipsis can occur only once at the end of a parameter list.

## Common Programming Error 7.5

Placing an ellipsis indicating a variable-length argument list in the middle of a parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest {
5     // calculate average
6     public static double average(double... numbers) {
7        double total = 0.0;
8
9        // calculate total using the enhanced for statement
10       for (double d : numbers) {
11          total += d;
12       }
13
14       return total / numbers.length;
15    }
16
```

**Fig. 7.20** | Using variable-length argument lists. (Part 1 of 3.)

```
17      public static void main(String[] args) {
18         double d1 = 10.0;
19         double d2 = 20.0;
20         double d3 = 30.0;
21         double d4 = 40.0;
22
23         System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
24            d1, d2, d3, d4);
25
26         System.out.printf("Average of d1 and d2 is %.1f%n",
27            average(d1, d2));
28         System.out.printf("Average of d1, d2 and d3 is %.1f%n",
29            average(d1, d2, d3));
30         System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
31            average(d1, d2, d3, d4));
32      }
33   }
```

**Fig. 7.20** | Using variable-length argument lists. (Part 2 of 3.)

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

**Fig. 7.20** | Using variable-length argument lists. (Part 3 of 3.)

# 7.14 Using Command-Line Arguments

- It's possible to pass arguments from the command line to an application via method `main`'s `String[]` parameter, which receives an array of `String`s.
- Command-line arguments that appear after the class name in the `java` command are received by `main` in the `String` array `args`.
- The number of command-line arguments is obtained by accessing the array's `length` attribute.
- Command-line arguments are separated by white space, not commas.

```
1  // Fig. 7.21: InitArray.java
2  // Initializing an array using command-line arguments.
3
4  public class InitArray {
5     public static void main(String[] args) {
6        // check number of command-line arguments
7        if (args.length != 3) {
8           System.out.printf(
9              "Error: Please re-enter the entire command, including%n" +
10             "an array size, initial value and increment.%n");
11       }
```

**Fig. 7.21** | Initializing an array using command-line arguments. (Part 1 of 4.)

```
12          else {
13             // get array size from first command-line argument
14             int arrayLength = Integer.parseInt(args[0]);
15             int[] array = new int[arrayLength];
16
17             // get initial value and increment from command-line arguments
18             int initialValue = Integer.parseInt(args[1]);
19             int increment = Integer.parseInt(args[2]);
20
21             // calculate value for each array element
22             for (int counter = 0; counter < array.length; counter++) {
23                array[counter] = initialValue + increment * counter;
24             }
25
26             System.out.printf("%s%8s%n", "Index", "Value");
27
28             // display array index and value
29             for (int counter = 0; counter < array.length; counter++) {
30                System.out.printf("%5d%8d%n", counter, array[counter]);
31             }
32          }
33       }
34    }
```

**Fig. 7.21** | Initializing an array using command-line arguments. (Part 2 of 4.)

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

**Fig. 7.21** | Initializing an array using command-line arguments. (Part 3 of 4.)

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

**Fig. 7.21** | Initializing an array using command-line arguments. (Part 4 of 4.)

# 7.15 Class `Arrays`

- `Arrays` class
  - Provides `static` methods for common array manipulations.
- Methods include
  - `sort` for sorting an array (ascending order by default)
  - `binarySearch` for searching a sorted array
  - `equals` for comparing arrays
  - `fill` for placing values into an array.
- Methods are overloaded for primitive-type arrays and for arrays of objects.
- `System` class `static` `arraycopy` method
  - Copies contents of one array into another.

```
1  // Fig. 7.22: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations {
6     public static void main(String[] args) {
7        // sort doubleArray into ascending order
8        double[] doubleArray = {8.4, 9.3, 0.2, 7.9, 3.4};
9        Arrays.sort(doubleArray);
10       System.out.printf("%ndoubleArray: ");
11
12       for (double value : doubleArray) {
13          System.out.printf("%.1f ", value);
14       }
15
```

**Fig. 7.22** | `Arrays` class methods and `System.arraycopy`. (Part 1 of 5.)

```
16          // fill 10-element array with 7s
17          int[] filledIntArray = new int[10];
18          Arrays.fill(filledIntArray, 7);
19          displayArray(filledIntArray, "filledIntArray");
20
21          // copy array intArray into array intArrayCopy
22          int[] intArray = {1, 2, 3, 4, 5, 6};
23          int[] intArrayCopy = new int[intArray.length];
24          System.arraycopy(intArray, 0, intArrayCopy, 0, intArray.length);
25          displayArray(intArray, "intArray");
26          displayArray(intArrayCopy, "intArrayCopy");
27
```

**Fig. 7.22** | `Arrays` class methods and `System.arraycopy`. (Part 2 of 5.)

```
28          // compare intArray and intArrayCopy for equality
29          boolean b = Arrays.equals(intArray, intArrayCopy);
30          System.out.printf("%n%nintArray %s intArrayCopy%n",
31             (b ? "==" : "!="));
32
33          // compare intArray and filledIntArray for equality
34          b = Arrays.equals(intArray, filledIntArray);
35          System.out.printf("intArray %s filledIntArray%n",
36             (b ? "==" : "!="));
37
38          // search intArray for the value 5
39          int location = Arrays.binarySearch(intArray, 5);
40
```

**Fig. 7.22** | `Arrays` class methods and `System.arraycopy`. (Part 3 of 5.)

```
41          if (location >= 0) {
42             System.out.printf(
43                "Found 5 at element %d in intArray%n", location);
44          }
45          else {
46             System.out.println("5 not found in intArray");
47          }
48
49          // search intArray for the value 8763
50          location = Arrays.binarySearch(intArray, 8763);
51
52          if (location >= 0) {
53             System.out.printf(
54                "Found 8763 at element %d in intArray%n", location);
55          }
56          else {
57             System.out.println("8763 not found in intArray");
58          }
59       }
```

**Fig. 7.22** | `Arrays` class methods and `System.arraycopy`. (Part 4 of 5.)

```
60
61      // output values in each array
62      public static void displayArray(int[] array, String description) {
63         System.out.printf("%n%s: ", description);
64
65         for (int value : array) {
66            System.out.printf("%d ", value);
67         }
68      }
69   }
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

**Fig. 7.22** | `Arrays` class methods and `System.arraycopy`. (Part 5 of 5.)

## Error-Prevention Tip 7.3

When comparing array contents, always use `Arrays.equals(array1, array2)`, which compares the two arrays' contents, rather than `array1.equals(array2)`, which compares whether `array1` and `array2` refer to the same array object.

## Common Programming Error 7.6

Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined.

# 7.15 Class Arrays

***Java SE 8—Class `Arrays` Method `parallelSort`***

- The `Arrays` class now has several new "parallel" methods that take advantage of multi-core hardware.
- `Arrays` method `parallelSort` can sort large arrays more efficiently on multi-core systems.
- In Section 23.12, we create a very large array and use features of the Java SE 8 Date/Time API to compare how long it takes to sort the array with methods `sort` and `parallelSort`.

# 7.16 Introduction to Collections and Class `ArrayList`

- Java API provides several predefined data structures, called collections, used to store groups of related objects in memory.
  - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
  - Reduce application-development time.
- Arrays do not automatically change their size at execution time to accommodate additional elements.
- `ArrayList<T>` (package `java.util`) can dynamically change its size to accommodate more elements.
  - `T` is a placeholder for the type of element stored in the collection.
- Classes with this kind of placeholder that can be used with any type are called generic classes.

| Method | Description |
|---|---|
| `add` | Overloaded to add an element to the *end* of the `ArrayList` or at a specific index in the `ArrayList`. |
| `clear` | Removes all the elements from the `ArrayList`. |
| `contains` | Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`. |
| `get` | Returns the element at the specified index. |
| `indexOf` | Returns the index of the first occurrence of the specified element in the `ArrayList`. |
| `remove` | Overloaded. Removes the first occurrence of the specified value or the element at the specified index. |
| `size` | Returns the number of elements stored in the `ArrayList`. |
| `trimToSize` | Trims the capacity of the `ArrayList` to the current number of elements. |

**Fig. 7.23** | Some methods of class `ArrayList<E>`.

# 7.16 Introduction to Collections and Class `ArrayList` (Cont.)

- Figure 7.24 demonstrates some common `ArrayList` capabilities.
- An `ArrayList`'s capacity indicates how many items it can hold without growing.
- When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
  - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
  - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

# 7.16 Introduction to Collections and Class `ArrayList` (Cont.)

- Method `add` adds elements to the `ArrayList`.
  - One-argument version appends its argument to the end of the `ArrayList`.
  - Two-argument version inserts a new element at the specified position.
  - Collection indices start at zero.
- Method `size` returns the number of elements in the `ArrayList`.
- Method `get` obtains the element at a specified index.
- Method `remove` deletes an element with a specific value.
  - An overloaded version of the method removes the element at the specified index.
- Method `contains` determines if an item is in the `ArrayList`.

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<E> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection {
6     public static void main(String[] args) {
7        // create a new ArrayList of Strings with an initial capacity of 10
8        ArrayList<String> items = new ArrayList<String>();
9
10       items.add("red"); // append an item to the list
11       items.add(0, "yellow"); // insert "yellow" at index 0
12
13       // header
14       System.out.print(
15          "Display list contents with counter-controlled loop:");
16
```

**Fig. 7.24** | Generic `ArrayList<E>` collection demonstration. (Part 1 of 4.)

```
17          // display the colors in the list
18          for (int i = 0; i < items.size(); i++) {
19             System.out.printf(" %s", items.get(i));
20          }
21
22          // display colors using enhanced for in the display method
23          display(items,
24             "%nDisplay list contents with enhanced for statement:");
25
26          items.add("green"); // add "green" to the end of the list
27          items.add("yellow"); // add "yellow" to the end of the list
28          display(items, "List with two new elements:");
29
30          items.remove("yellow"); // remove the first "yellow"
31          display(items, "Remove first instance of yellow:");
32
33          items.remove(1); // remove item at index 1
34          display(items, "Remove second list element (green):");
```

**Fig. 7.24** | Generic `ArrayList<E>` collection demonstration. (Part 2 of 4.)

```
35
36          // check if a value is in the List
37          System.out.printf("\"red\" is %sin the list%n",
38             items.contains("red") ? "": "not ");
39
40          // display number of elements in the List
41          System.out.printf("Size: %s%n", items.size());
42       }
43
44       // display the ArrayList's elements on the console
45       public static void display(ArrayList<String> items, String header) {
46          System.out.printf(header); // display header
47
48          // display each element in items
49          for (String item : items) {
50             System.out.printf(" %s", item);
51          }
52
53          System.out.println();
54       }
55    }
```

**Fig. 7.24** | Generic `ArrayList<E>` collection demonstration. (Part 3 of 4.)

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

**Fig. 7.24** | Generic `ArrayList<E>` collection demonstration. (Part 4 of 4.)

# 7.16 Introduction to Collections and Class `ArrayList` (Cont.)

***Java SE 7—Diamond (<>) Notation for Creating an Object of a Generic Class***

- Consider this statement from Fig. 7.24:
  - `ArrayList<String> items = new ArrayList<String>();`
- Notice that `ArrayList<String>` appears in the variable declaration and in the class instance creation expression. Java SE 7 introduced the diamond (<>) notation to simplify statements like this. Using <> in a class instance creation expression for an object of a *generic* class tells the compiler to determine what belongs in the angle brackets.

# 7.16 Introduction to Collections and Class `ArrayList` (Cont.)

- In Java SE 7 and higher, the preceding statement can be written as:
  - `ArrayList<String> items = new ArrayList<>();`
- When the compiler encounters the diamond (`<>`) in the class instance creation expression, it uses the declaration of variable items to determine the `ArrayList`'s element type (`String`)—this is known as *inferring the element type*.

# 7.17 (Optional) GUI and Graphics Case Study: Drawing Arcs

```
1  // Fig. 7.25: DrawRainbowController.java
2  // Drawing a rainbow using arcs.
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.canvas.Canvas;
6  import javafx.scene.canvas.GraphicsContext;
7  import javafx.scene.paint.Color;
8  import javafx.scene.shape.ArcType;
9
10 public class DrawRainbowController {
11    @FXML private Canvas canvas;
12
13    // colors to use in the rainbow, starting from the innermost
14    // The two white entries result in an empty arc in the center
15    private final Color[] colors = {
16       Color.WHITE, Color.WHITE, Color.VIOLET, Color.INDIGO, Color.BLUE,
17       Color.GREEN, Color.YELLOW, Color.ORANGE, Color.RED};
18
```

**Fig. 7.25** | Drawing a rainbow using arcs. (Part 1 of 3.)

```
19      // draws a rainbow using arcs
20      @FXML
21      void drawRainbowButtonPressed(ActionEvent event) {
22         // get the GraphicsContext, which is used to draw on the Canvas
23         GraphicsContext gc = canvas.getGraphicsContext2D();
24
25         final int radius = 20; // radius of an arc
26
27         // draw the rainbow near the bottom-center
28         final double centerX = canvas.getWidth() / 2;
29         final double maxY = canvas.getHeight() - 10;
30
31         // draws filled arcs starting with the outermost
32         for (int counter = colors.length; counter > 0; counter--) {
33            // set the color for the current arc
34            gc.setFill(colors[counter - 1]);
35
36            // fill the arc from 0 to 180 degrees
37            gc.fillArc(centerX - counter * radius,
38               maxY - counter * radius, counter * radius * 2,
39               counter * radius * 2, 0, 180, ArcType.OPEN);
40         }
41      }
42   }
```

**Fig. 7.25** | Drawing a rainbow using arcs. (Part 2 of 3.)


**Fig. 7.25** | Drawing a rainbow using arcs. (Part 3 of 3.)

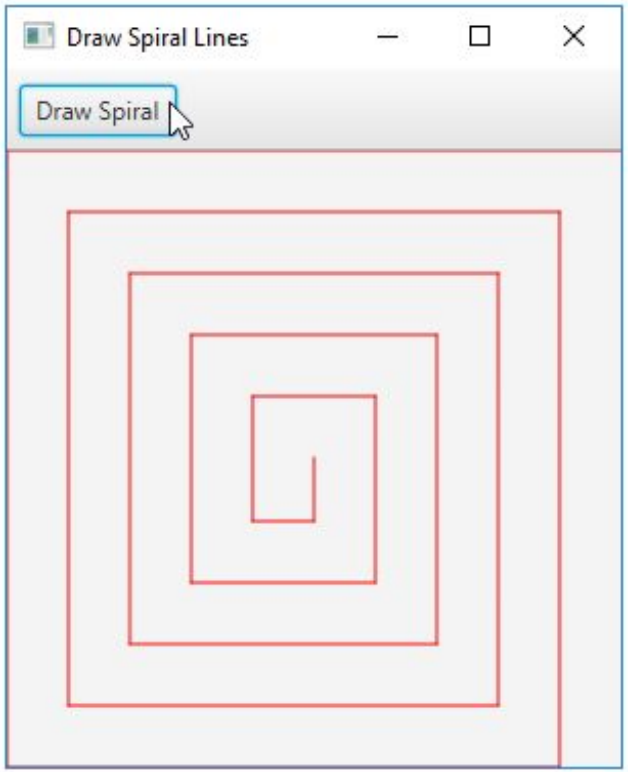


**Fig. 7.26** | Drawing a spiral using `strokeLine`.

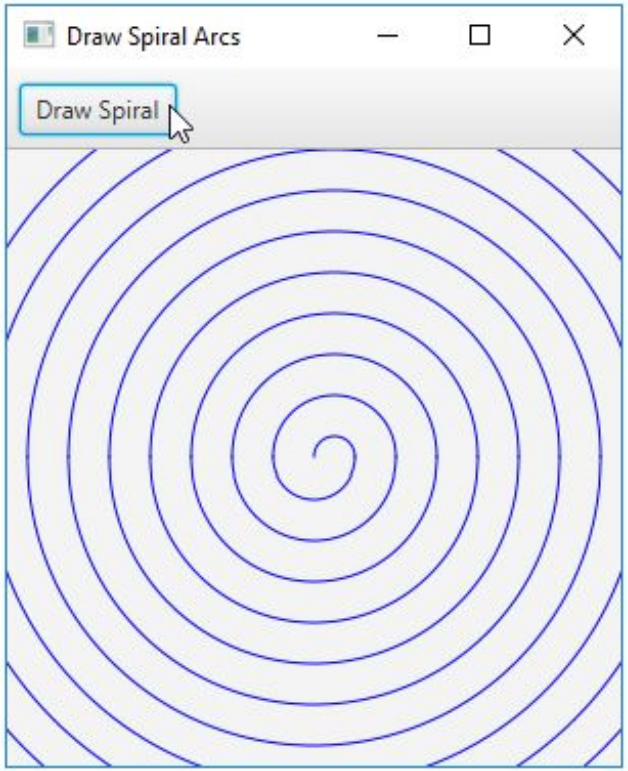


**Fig. 7.27** | Drawing a spiral using `strokeArc`.