School of Science & Engineering
Department of CSE

Lecture-6: Arrays

Semester: Summer 2024

Object-Oriented Problem Solving

Arrays

Based on Chapters 7 & 8 of "Introduction to Java Programming" by Y. Daniel Liang.

Outline

- Introduction (7.1)
- Array Basics - Declaring Arrays (7.2.1)
- Array Basics - Creating Arrays (7.2.2)
- Array Basics - Array Size and Default Values (7.2.3)
- Array Basics – Accessing Array Elements (7.2.4)
- Array Basics - Array Initializers (7.2.5)
- Array Bascis - Processing Arrays (7.2.6)
- Array Basics - Foreach Loops (7.2.7)
- Copying Arrays (7.5)
- Passing Arrays to Methods (7.6)
- Returning an Array from a Method (7.7)
- Variable-Length Argument Lists (7.9)
- Command-Line Arguments (7.13)
- Two-Dimensional Arrays Basics (8.2)
- Processing Two-Dimensional Arrays (8.3)

Introduction

- An *array* is a data structure which stores a fixed-size sequential collection of elements of the same type.
- A single array variable can reference a large collection of data.

Array Basics
Declaring Arrays

- To use an array in a program, you must *declare* a variable to reference the array and specify the array's elements type.
- The syntax for declaring an array variable is:

  elementType [] arrayRefVar;
- The *elementType* can be any data type.
  - All elements in the array will have the same data type.
- Example:

  double [] myList;

Array Basics
Declaring Arrays (Cont.)

- Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array.
  - It creates only a storage location for the reference to an array.
  - If a variable does not contain a reference to an array, the value of the variable is *null*.

Array Basics
Creating Arrays

- An array is created using the *new* operator with the following syntax:

  arrayRefVar = new elementType [arraySize];

  - This statement does two things:
    - It creates an array using new elementType [arraySize]
    - It assigns the reference of the newly created array to the variable arrayRefVar.
- Example:

```
double [] myList;                    //array declaration
mylist = new double [10];                    //array creation
```

Array Basics
Creating Arrays (Cont.)

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:

  elementType [] arrayRefVar = new elementType [arraySize];

- Example:

  double [] myList = new double [10];

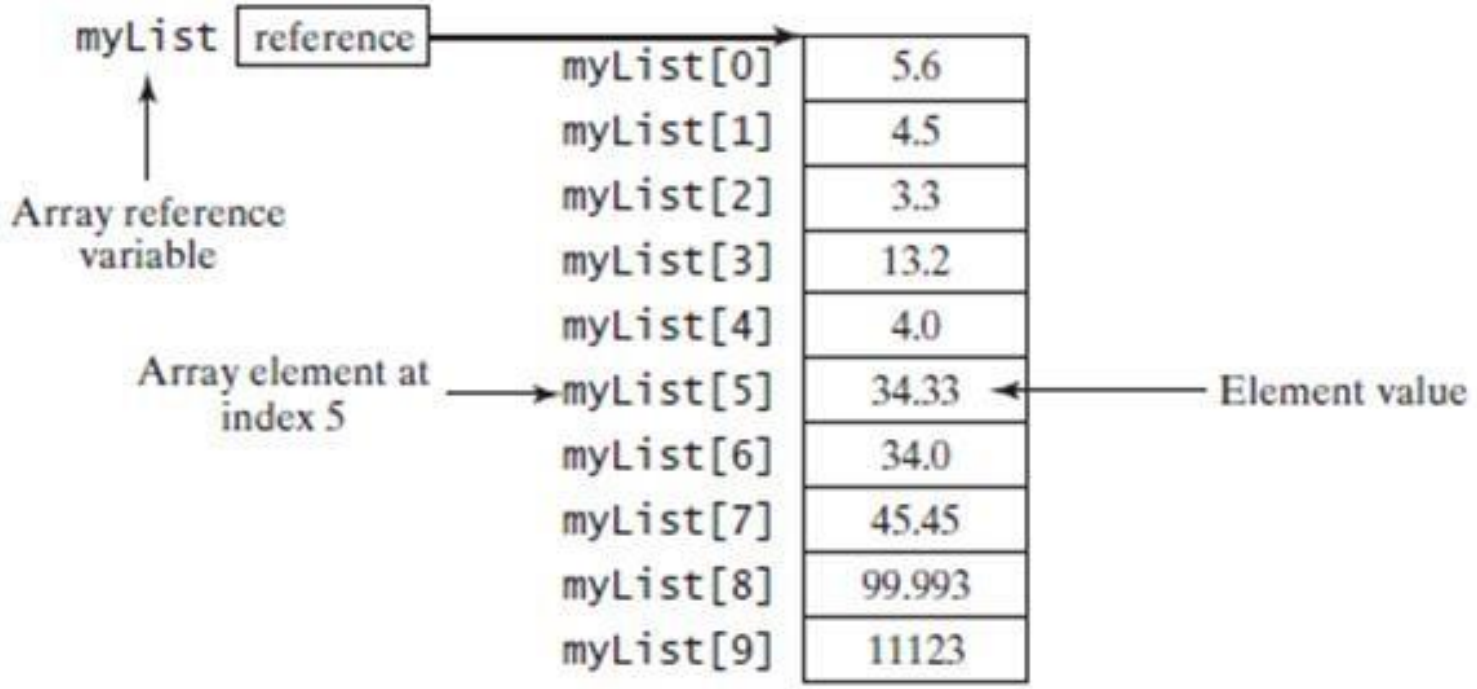
  – This statement declares an array variable, *myList*, creates an array of ten elements of *double* type, and assigns its reference to *myList*.

Array Basics
Creating Arrays (Cont.)

- The syntax to assign values to array elements:
  arrayRefVar [index] = value;
- Example:

```
double [] myList = new double [10];
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
        .
        .
myList[9] = 11123;
```

Array Basics
Creating Arrays (Cont.)



- An array variable that appears to hold an array actually contains a reference to that array.
- Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored.

Array Basics
Array Size and Default Values

- When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it.
- The size of an array cannot be changed after the array is created.
- Array size can be obtained using: *arrayRefVar.length*
- When an array is created, its elements are assigned their default values:
  - *0* for *numeric* primitive data types.
  - *\u0000* for *char* types.
  - *False* for *boolean* types.

Array Basics
Accessing Array Elements

- The array elements are accessed through the index.
- Array indices are 0 based.
  - They range from *0* to *arrayRefVar.length-1*.
- Each element in the array is represented using the following syntax:

  arrayRefVar [index]

Array Basics
Accessing Array Elements (Cont.)

- An indexed variable can be used in the same way as a regular variable.
- Examples:

```
myList[2] = myList[0] + myList[1];
for (int i=0; i < myList.length; i++){
        myList[i] = i;
}
```

Array Basics
Array Initializers

- *Array initializer* is a shorthand notation which combines the *declaration*, *creation*, and *initialization* of an array in one statement.
- The syntax for array initializer:

  elementType [] arrayRefVar = {value0, value1, …., valuek};
- Example:

  double [] myList = {1.9, 2.5, 3.4, 4.5};
- Using an array initializer, you have to declare, create, and initialize the array all in one statement.
  - Splitting it would cause a syntax error.

  double[] myList;

  myList = {1.9, 2.9, 3.4, 3.5}; // causes a syntax error

Array Basics
Processing Arrays

- When processing array elements, you will often use a *for* loop.
  - All elements in an array are of the same type and they are evenly processed in the same fashion repeatedly using a loop.
  - Since the size of the array is known, it is natural to use a *for* loop.

Array Basics
Processing Arrays (Examples)

- *Displaying arrays*: to print an array, you have to print each element in the array using a loop like the following:

```
For (int i = 0; i < myList.length; i++)
    System.out.print (myList[i] + " ");
```

  - *Note:* For an array of the *char[]* type, it can be printed using one print statement

```
char[] city = {'A', 'm', 'm', 'a', 'n'};
System.out.println(city);
```

- *Summing all elements:* Use a variable named total to store the sum. Initially total is 0. Add each element in the array to total using a loop like this:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
      total += myList[i];
}
```

```
public class MyListExample {
    public static void main(String[] args) {
        // Creating an array of doubles
        double[] myList = {3.14, 1.618, 2.718, 1.414};

        // Iterating over the array and printing each element with
a space
        for (int i = 0; i < myList.length; i++) {
            System.out.print(myList[i] + " ");
        }
    }
}
```

```
public class MyListSum {
    public static void main(String[] args) {
        // Creating an array of doubles
        double[] myList = {3.14, 1.618, 2.718, 1.414};

        // Initializing a variable to hold the total sum
        double total = 0;

        // Iterating over the array and summing up all elements
        for (int i = 0; i < myList.length; i++) {
            total += myList[i];
        }

        // Printing the total sum
        System.out.println("Total sum: " + total);
    }
}
```

Array Basics
Foreach loops

- Java supports a convenient *for* loop, known as a *foreach loop*, which enables you to traverse the array sequentially without using an index variable.
- For example, the following code displays all the elements in the array *myList*:

```
for (double e: myList) {
    System.out.println(e);
}
```

  - You can read the code as "for each element *e* in *myList*, do the following."
  - Note that the variable, *e*, must be declared as the same type as the elements in *myList*.
- In general, the syntax for a *foreach* loop is:

```
for (elementType element: arrayRefVar) {
    // Process the element
}
```

- You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

```
import java.util.ArrayList;

public class MyListElements {
    public static void main(String[] args) {
        // Creating a list of doubles
        ArrayList<Double> myList = new ArrayList<Double>();

        // Adding some elements to the list
        myList.add(3.14);
        myList.add(1.618);
        myList.add(2.718);
        myList.add(1.414);

        // Iterating over the list and printing each element
        for (double e : myList) {
            System.out.println(e);
        }
    }
}
```

7.3 Case Study: Analyzing Numbers

The problem is to write a program that finds the number of items above the average of all items.

Now you can write a program using arrays to solve the problem proposed at the beginning of this chapter. The problem is to read 100 numbers, get the average of these numbers, and find the number of the items greater than the average. To be flexible for handling any number of input, we will let the user enter the number of input, rather than fixing it to 100. Listing 7.1 gives a solution.

```
// LISTING 7.1 AnalyzeNumbers.java
public class AnalyzeNumbers {
public static void main(String[] args) { //numbers[0]
java.util.Scanner input = new java.util.Scanner(System.in); // numbers[1]:
System.out.print("Enter the number of items: "); // numbers[2]:
int n = input.nextInt();
double [] numbers = new double[n];
double sum = 0;

// numbers[i]:
System.out.print("Enter the numbers: "); // store number in array
for (int i = 0; i < n; i++) {          // numbers[n - 3]:
numbers[i] = input.nextDouble();       // numbers[n - 2]:
sum += numbers[i];                     // numbers[n - 1]:
 }

double average = sum / n;                          // get average

int count = 0; // The number of elements above average
for (int i = 0; i < n; i++)
if (numbers[i] > average)                          // above average?
count++;

System.out.println("Average is " + average);
System.out.println("Number of elements above the average is " + count);
}
}
```

7.4 Case Study: Deck of Cards

The problem is to create a program that will randomly select four cards from a deck of cards. Say you want to write a program that will pick four cards at random from a deck of **52** cards. All the cards can be represented using an array named **deck**, filled with initial values **0** to **51**, as follows:

```
int[] deck = new int[52];
// Initialize cards
for (int i = 0; i < deck.length; i++)
deck[i] = i;
```

Card numbers **0** to **12**, **13** to **25**, **26** to **38**, and **39** to **51** represent 13 Spades, 13 Hearts, 13 Diamonds, and 13 Clubs, respectively, as shown in Figure 7.2.

cardNumber / 13 determines the suit of the card and **cardNumber % 13** determines the rank of the card, as shown in Figure 7.3. After shuffling the array **deck**, pick the first four cards from **deck**. The program displays the cards from these four card numbers.

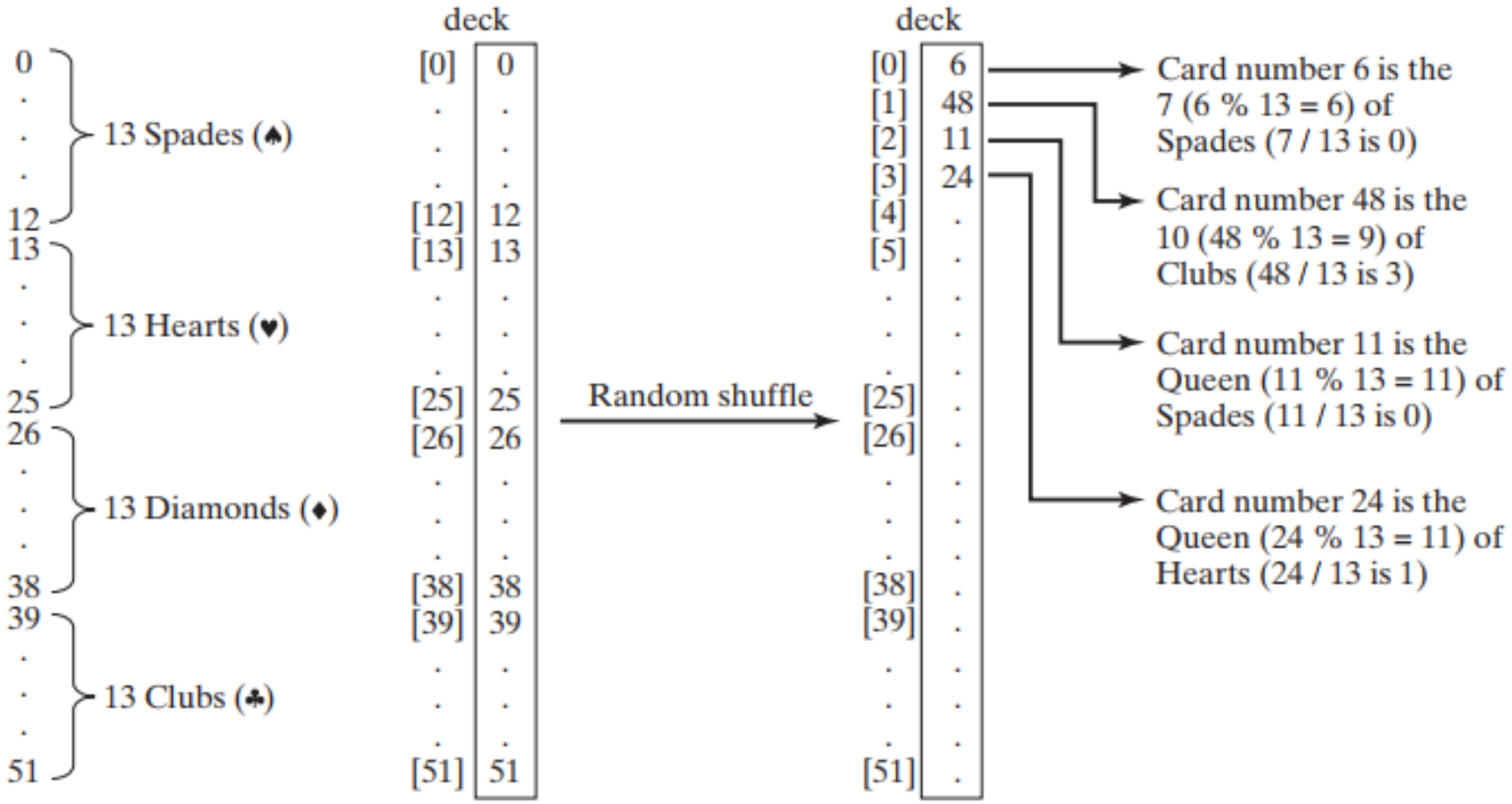


FIGURE 7.2 52 cards are stored in an array named **deck**.

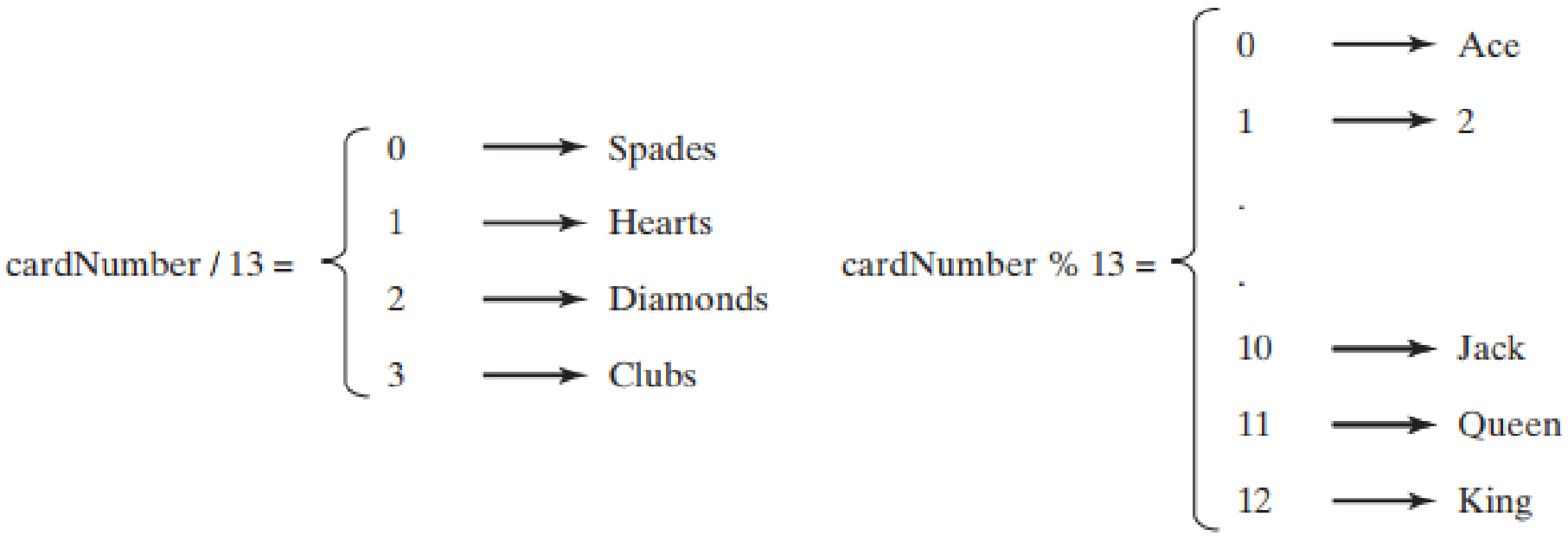
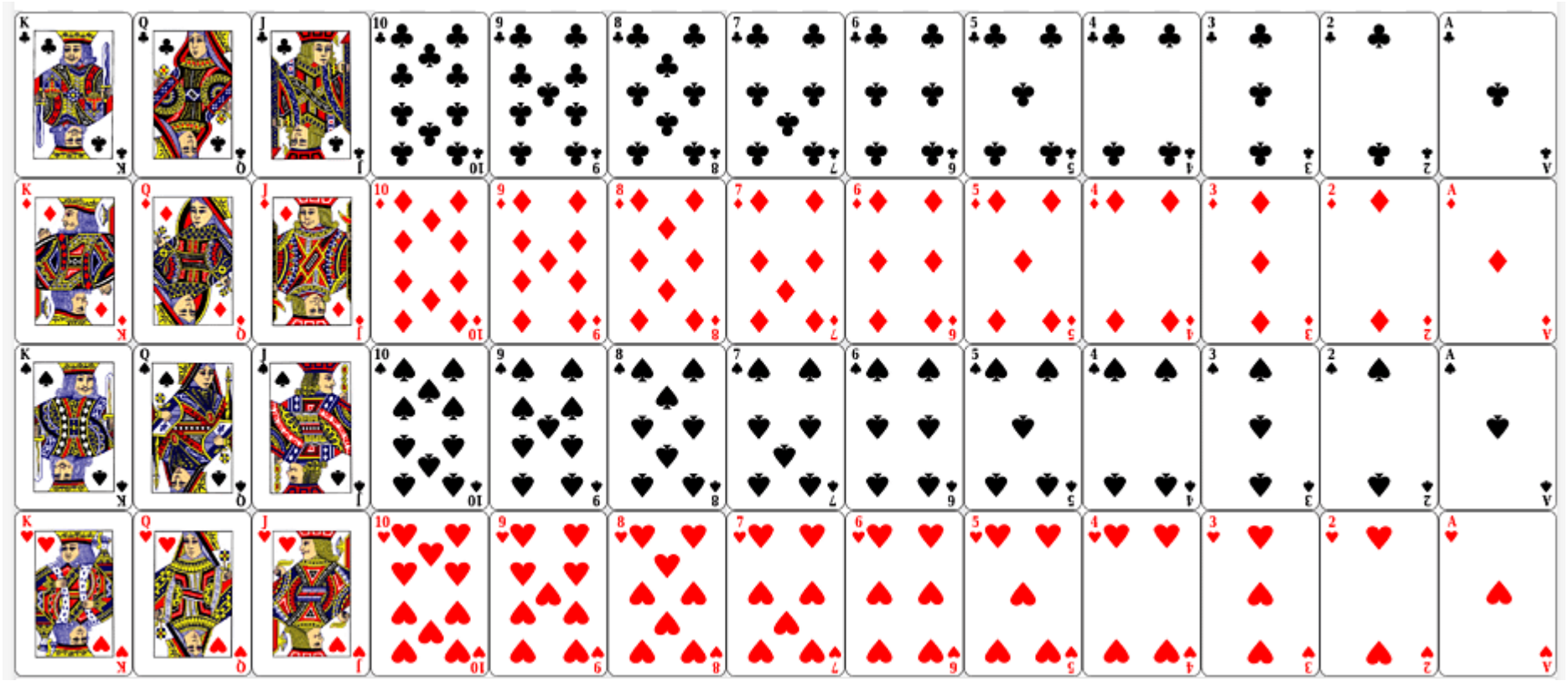


FIGURE 7.3 `CardNumber` identifies a card's suit and rank number.



```
// LISTING 7.2 DeckOfCards.java
public class DeckOfCards {
public static void main(String[] args) {

int[] deck = new int[52];                    // create array deck
String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"}; // array of strings
String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9","10", "Jack", "Queen", "King"}; // array of strings

// Initialize the cards
for (int i = 0; i < deck.length; i++) // initialize deck
deck[i] = i;

// Shuffle the cards
for (int i = 0; i < deck.length; i++) { // shuffle deck
// Generate an index randomly
int index = (int)(Math.random() * deck.length);
int temp = deck[i];
deck[i] = deck[index];
deck[index] = temp;
 }

// Display the first four cards
for (int i = 0; i < 4; i++) {
String suit = suits[deck[i] / 13]; // suit of a card
String rank = ranks[deck[i] % 13]; // rank of a card
System.out.println("Card number " + deck[i] + ": " + rank + " of " + suit);
}
}
}
```

Note

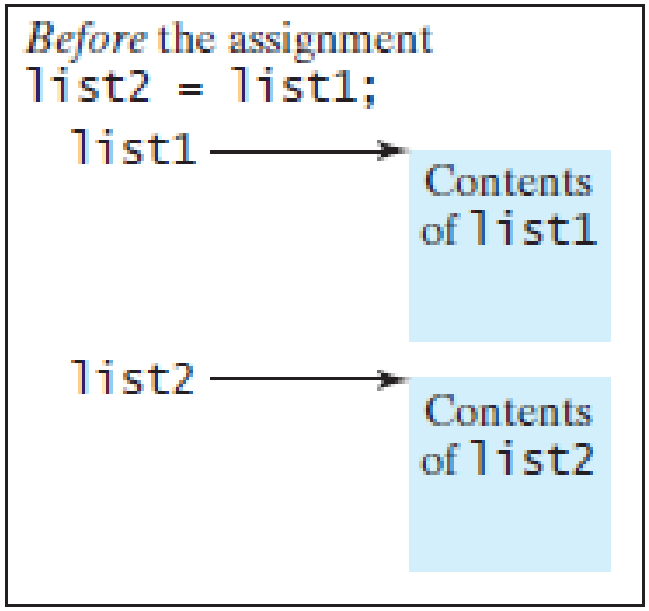
- Accessing an array out of bounds is a common programming error that throws a runtime *ArrayIndexOutOfBoundsException*.
- To avoid it, make sure that you do not use an index beyond **arrayRefVar.length – 1**.
- Programmers often mistakenly reference the first element in an array with index **1**, but it should be **0**.
  - This is called the *off-by-one error*.
- Another common off-by-one error in a loop is using **<=** where **<** should be used.
- For example, the following loop is wrong.

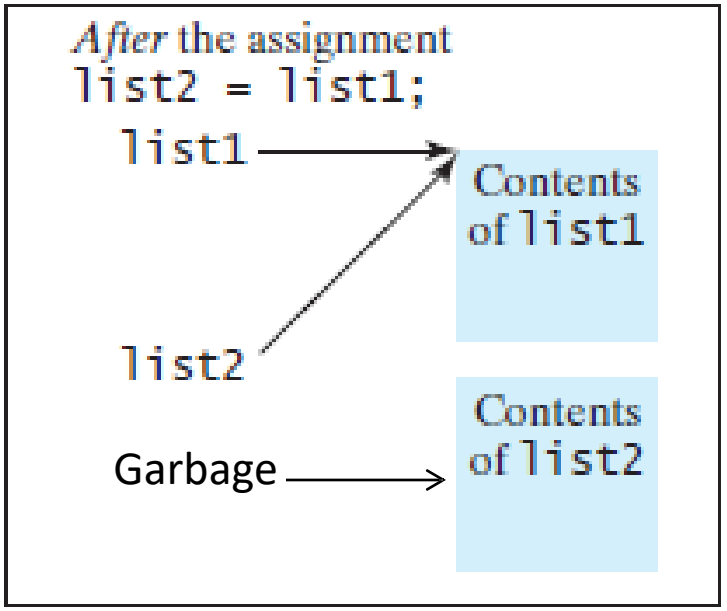
```
for (int i = 0; i <= list.length; i++)
    System.out.print(list[i] + " ");
```

  - The **<=** should be replaced by **<**.

Copying Arrays

- The assignment operator does not copy the contents of an array into another, it instead merely copies the reference values.





Copying Arrays (Cont.)

- To copy the contents of one array into another, you have to copy the *array's individual elements* into the other array.
- Use a loop to copy every element from the source array to the corresponding element in the target array.
- Example:

```
int [] sourceArray = {2, 3, 1, 5, 10};
int [] targetArray = new int [sourceArray.length];
for (int i=0; i < sourceArray.length; i++)
        targetArray [i] = sourceArray [i];
```

```
public class ArrayCopyExample {
    public static void main(String[] args) {
        // Initializing the source array with some integer values
        int[] sourceArray = {2, 3, 1, 5, 10};

        // Creating a target array with the same length as the source array
        int[] targetArray = new int[sourceArray.length];

        // Copying elements from sourceArray to targetArray
        for (int i = 0; i < sourceArray.length; i++) {
            targetArray[i] = sourceArray[i];
        }
        // Printing the elements of targetArray to verify the copy
        System.out.println("Elements of targetArray:");
        for (int i = 0; i < targetArray.length; i++) {
            System.out.print(targetArray[i] + " ");
        }
    }
}
```

Copying Arrays (Cont.)

- Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (**=**), as follows:
  - *list2 = list1;*

Passing Arrays to Methods

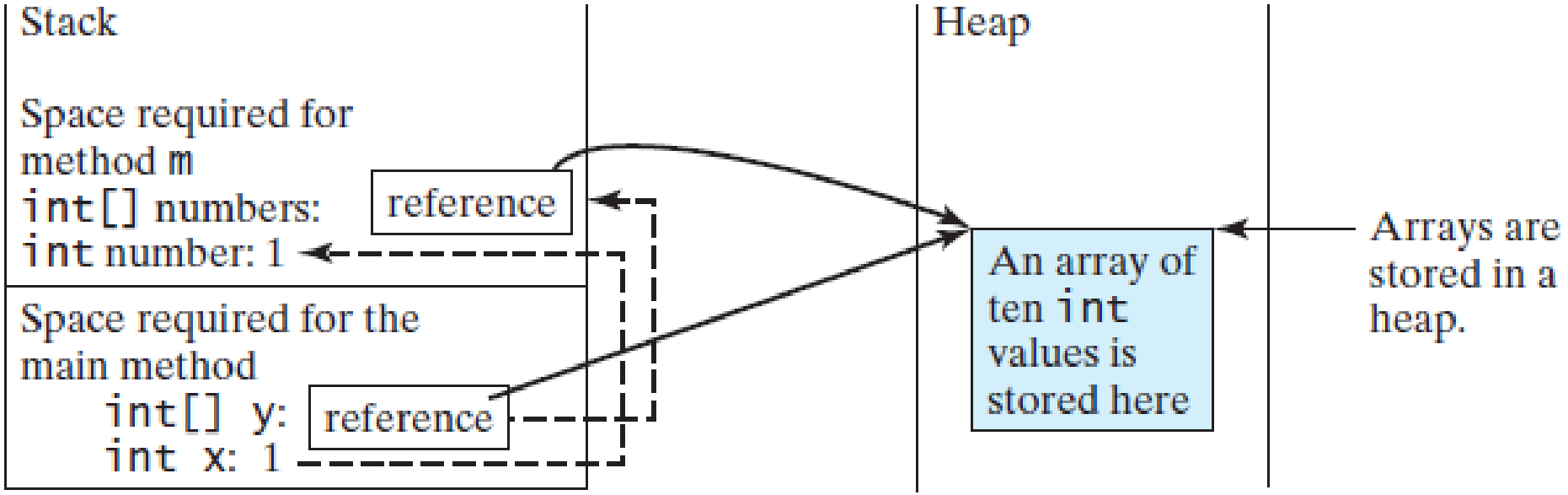
- When passing an array to a method, the *reference* of the array is passed to a method.
- This differs from passing arguments of a primitive type:
  - For an argument of a primitive type, the argument's value is passed.
    - *The passed variable will not be affected by any change to the value inside the method.*
  - For any argument of an array type, the value of the argument is a reference to an array.
    - *The passed array will be affected by any change inside the method.*

Passing Arrays to Methods: Example

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

```
x is 1
y[0] is 5555
```

Passing Arrays to Methods: Example (Cont.)



Returning an Array from a Method

- When a method returns an array, the reference of the array is returned.
- Example:

```
public static int[] copy(int [] list){
        int [] result = new int [list.length];
        for (int i=0; i < list.length; i++)
                result[i]=list[i];
        return result;
}
```

Example of this method invocation:
int [] list1 = {1, 2, 3, 4, 5};
int [] list2 = copy(list1);

```
public class ArrayCopyMethod {

   // Method to copy an array
   public static int[] copy(int[] list) {
      // Create a new array to hold the copied elements
      int[] result = new int[list.length];
      // Copy each element from the original list to the new array
      for (int i = 0; i < list.length; i++) {
         result[i] = list[i];
      }
      // Return the new array
      return result;
   }
   public static void main(String[] args) {
      // Initialize the original array
      int[] originalArray = {2, 3, 1, 5, 10};
      // Call the copy method to create a copy of the original array
      int[] copiedArray = copy(originalArray);
      // Print the elements of the copied array to verify the copy
      System.out.println("Elements of copiedArray:");
      for (int i = 0; i < copiedArray.length; i++) {
         System.out.print(copiedArray[i] + " ");
      }
   }
}
```

Variable Length Argument List

- A variable number of arguments of the same type can be passed to a method and treated as an array.
- The parameter in the method is declared as follows:

  typeName... parameterName

- In the method declaration, you specify the type followed by an ellipsis (**...**).
- Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter.
  - Any regular parameters must precede it.

Variable Length Argument List (Cont.)

- Java treats a variable-length parameter as an array.
- You can pass an array or a variable number of arguments to a variable-length parameter.
- When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it.

VarArgsDemo.java

```
1  public class VarArgsDemo {
2    public static void main(String[] args) {
3      printMax(34, 3, 3, 2, 56.5);
4      printMax(new double[]{1, 2, 3});
5    }
6
7    public static void printMax(double... numbers) {
8      if (numbers.length == 0) {
9        System.out.println("No argument passed");
10       return;
11     }
12
13     double result = numbers[0];
14
15     for (int i = 1; i < numbers.length; i++)
16       if (numbers[i] > result)
17         result = numbers[i];
18
19     System.out.println("The max value is " + result);
20   }
21 }
```

Command-Line Arguments

- The *main* method has the parameter *args* of *String[]* type.
  - It is clear that *args* is an array of strings.
- You can pass strings to a main method from the command line when you run the program.
- The following command line, for example, starts a program named TestMain with three strings: arg0, arg1, and arg2:
  - *java TestMain arg0 arg1 arg2*
  - They don't have to appear in double quotes on the command line.
  - The strings are separated by a space.
- A string that contains a space must be enclosed in double quotes.
  - *java TestMain "First num" alpha 53*
  - Note that 53 is actually treated as a string.

Command-Line Arguments (Cont.)

- When the *main* method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to *args*.
- For example, if you invoke a program with *n* arguments, the Java interpreter creates an array like this one:
  - *args = new String[n];*

Two-Dimensional Arrays

- Two dimensional arrays are used to represent data in a matrix or a table.
- The syntax for declaring and creating two dimensional arrays is:

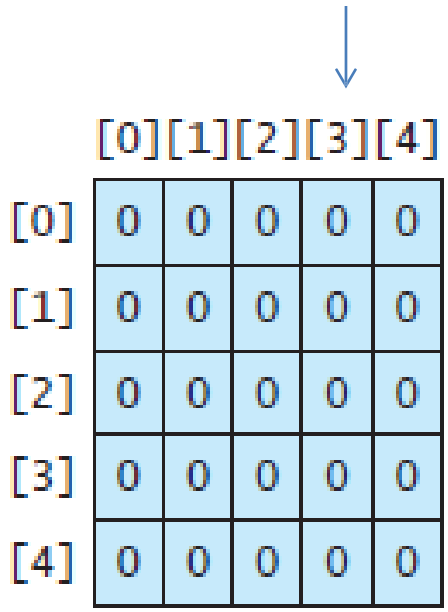
  elementType [] [] arrayRefVar;

  arrayRefVar = new elementType [numRows][numCols];
- An element in a two-dimensional array is accessed through a row and column index:

  arrayRefVar [rowIndex][colIndex];

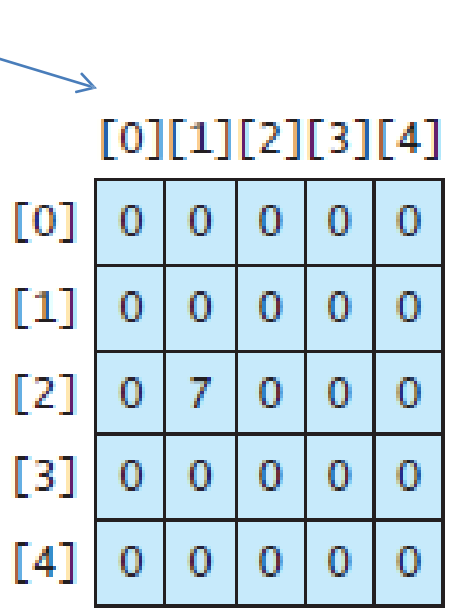
Two-Dimensional Arrays: Examples

int [][] matrix;



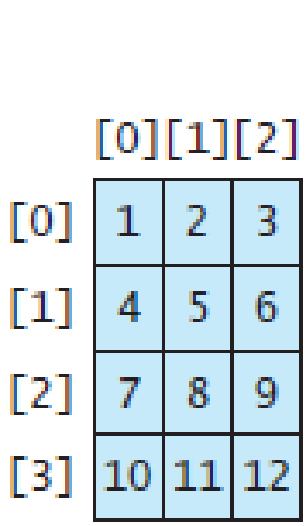
	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix = new int[5][5];
```



	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix[2][1] = 7;
```

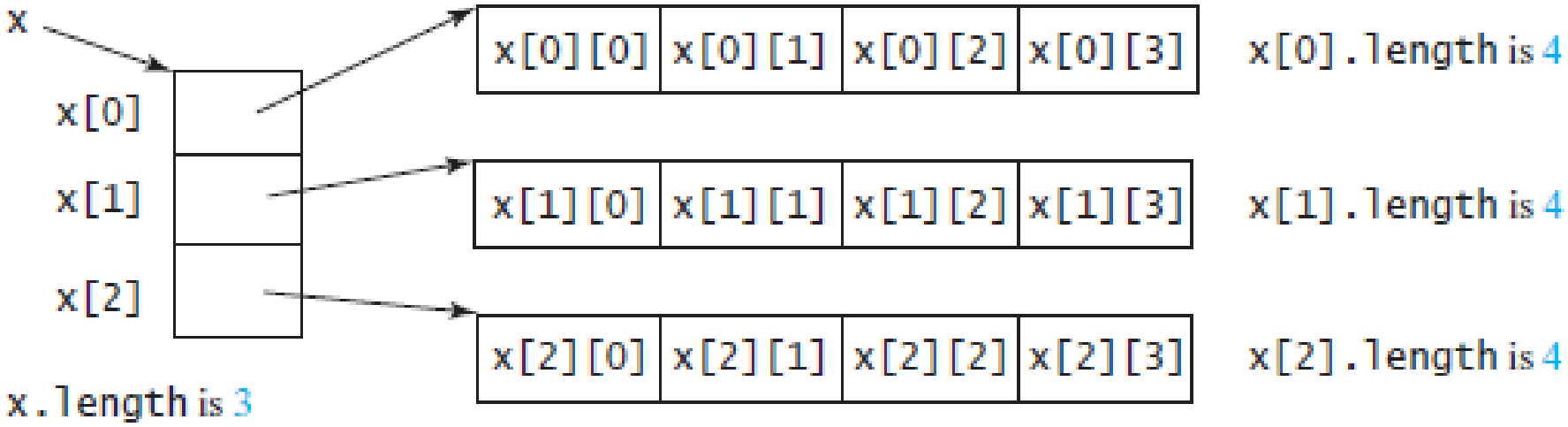


	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

```
int[][] array = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9},
   {10, 11, 12}
};
```

Two-Dimensional Arrays (Cont.)

- A two-dimensional array is actually an array in which each element is a one-dimensional array.



A simple Java program that demonstrates working with a two-dimensional array:

```
public class TwoDimensionalArrayExample {

    public static void main(String[] args) {
        // Create a 2D array (matrix) with 3 rows and 4 columns
        int[][] matrix = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        };

        // Print the 2D array
        System.out.println("The 2D array is:");
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println(); // Move to the next line after each row
        }
    }
}
```

Explanation:

1. 2D Array Declaration and Initialization:
   - The 2D array `matrix` is created with 3 rows and 4 columns.
   - Each row contains four integers.

2. Nested Loops for Traversing:
   - The outer loop (`for (int i = 0; i < matrix.length; i++)`) iterates through each row of the matrix.
   - The inner loop (`for (int j = 0; j < matrix[i].length; j++)`) iterates through each column in the current row.

3. Printing the Matrix:
   - `System.out.print(matrix[i][j] + " ");` prints each element in the row followed by a space.
   - `System.out.println();` moves to the next line after printing all the elements of a row, ensuring each row is displayed on a new line.

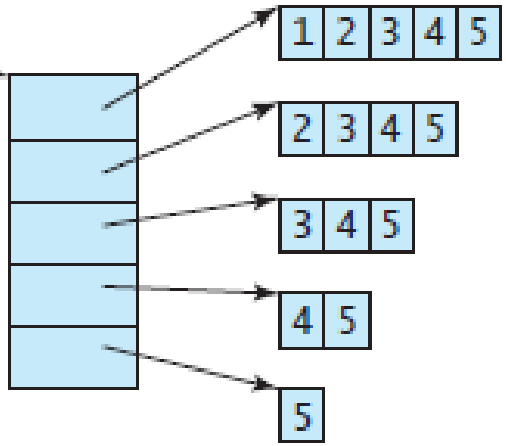
Output:

```
The 2D array is:
1 2 3 4
5 6 7 8
9 10 11 12
```

Ragged Arrays

- Each row in a two-dimensional array is itself an array.
- Thus, the rows can have different lengths.
  - An array of this kind is known as a *ragged array*.

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



Ragged Arrays (Cont.)

- You can also create a two-dimensional array if you know the sizes of its rows but do not know the values, using the following format:

```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

```
public class RaggedArrayExample {
    public static void main(String[] args) {
        // Declare a 2D ragged array with 4 rows
        int[][] raggedArray = new int[4][];

        // Initialize each row with different lengths
        raggedArray[0] = new int[]{1, 2, 3};
        raggedArray[1] = new int[]{4, 5};
        raggedArray[2] = new int[]{6, 7, 8, 9};
        raggedArray[3] = new int[]{10};

// Print the ragged array
        System.out.println("Contents of the ragged array:");
        for (int i = 0; i < raggedArray.length; i++) {
            System.out.print("Row " + i + ": ");
            for (int j = 0; j < raggedArray[i].length; j++) {
                System.out.print(raggedArray[i][j] + " ");
            }
            System.out.println(); // Move to the next line after each row
        }
    }
}
```

Processing Two-Dimensional Arrays

- Nested for loops are often used to process a two-dimensional array
- Suppose an array *matrix* is created as follows:
  - *int[][] matrix = new int[10][10];*
- To print the elements of the array *matrix* each row on a line:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    System.out.print(matrix[row][column] + " ");
  }

  System.out.println();
}
```

7.10 Searching Arrays

If an array is sorted, binary search is more efficient than linear search for finding an element in the array. *Searching* is the process of looking for a specific element in an array—for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, *linear search* and *binary search*.

7.10.1 The Linear Search Approach

The linear search approach compares the key element **key** sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns **-1**. The **linearSearch** method in Listing 7.6 gives the solution.

```java
// LISTING 7.6 LinearSearch.java
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
      if (key == list[i]) {
        return i; // return the index of the found key
      }
    }
    return -1; // return -1 if the key is not found
  }

  public static void main(String[] args) {
    int[] numbers = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}; // Sample array
    int key = 10; // The value we're searching for

    int result = linearSearch(numbers, key);

    if (result != -1) {
      System.out.println("The key " + key + " is found at index: " + result);
    } else {
      System.out.println("The key " + key + " is not found in the list.");
    }
  }
}
```

7.10.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array.

Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key
only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the
key only in the second half of the array.

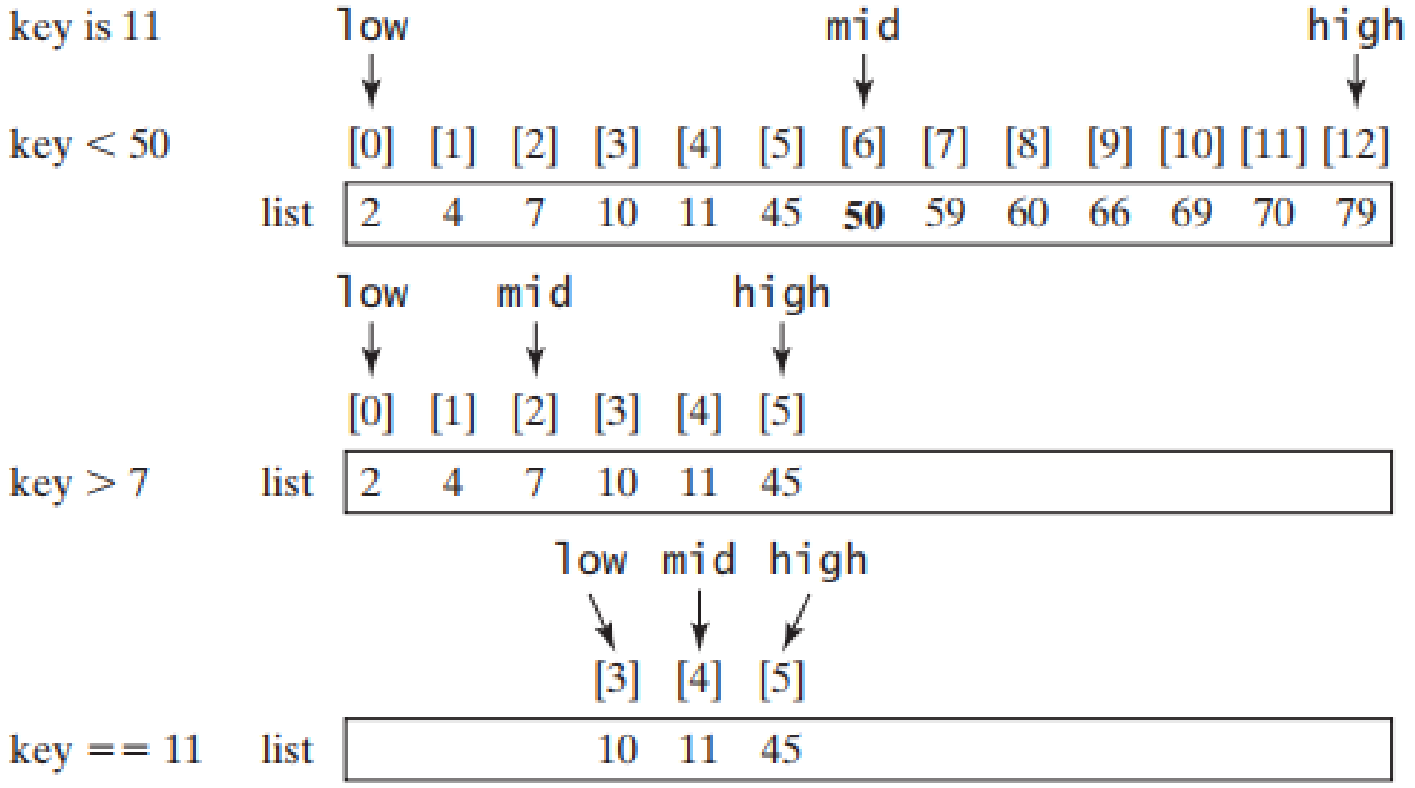


FIGURE 7.9 Binary search eliminates half of the list from further consideration after each comparison.

// LISTING 7.7 BinarySearch.java

```
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2; // Find the midpoint
      if (key < list[mid]) {
        high = mid - 1; // Search the left half
      } else if (key == list[mid]) {
        return mid; // Key found, return its index
      } else {
        low = mid + 1; // Search the right half
      }
    }

    return -low - 1; // Key not found, return a negative value
  }

  public static void main(String[] args) {
    int[] sortedNumbers = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}; // Sorted array
    int key = 10; // The value we're searching for

    int result = binarySearch(sortedNumbers, key);

    if (result >= 0) {
      System.out.println("The key " + key + " is found at index: " + result);
    } else {
      System.out.println("The key " + key + " is not found. Insert at index: " + (-result - 1));
    }
  }
}
```

7.13.2 Case Study: Calculator

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives an expression in one string argument. The expression consists of an integer followed by an operator and another integer. For example, to add two integers, use this command:

java Calculator 2 + 3

The program will display the following output:

2 + 3 = 5

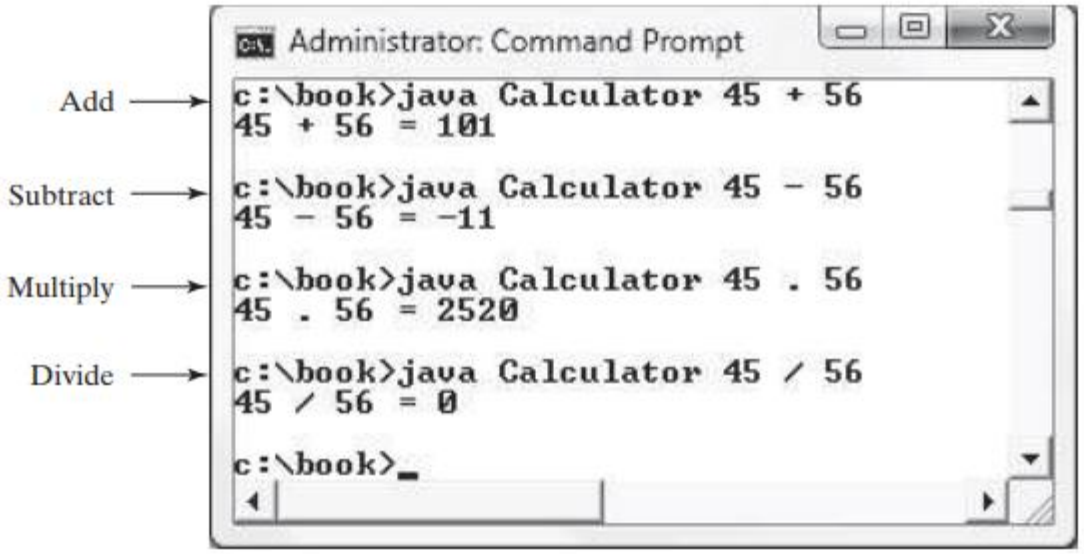


FIGURE 7.12 The program takes three arguments (`operand1 operator operand2`) from the command line and displays the expression and the result of the arithmetic operation.

```
// LISTING 7.9 Calculator.java
public class Calculator {
/** Main method */
public static void main(String[] args) {
// Check number of strings passed
if (args.length != 3) {
System.out.println("Usage: java Calculator operand1 operator operand2");
System.exit(0);
}

// The result of the operation
int result = 0;

// Determine the operator
switch (args[1].charAt(0)) {
case '+': result = Integer.parseInt(args[0]) + Integer.parseInt(args[2]);
break;
case '-': result = Integer.parseInt(args[0]) - Integer.parseInt(args[2]);
break;
case '.': result = Integer.parseInt(args[0]) * Integer.parseInt(args[2]);
break;
case '/': result = Integer.parseInt(args[0]) / Integer.parseInt(args[2]);
}

// Display result
System.out.println(args[0] + ' ' + args[1] + ' ' + args[2] + " = " + result);
}
}
```