CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-8: Arrays, Constructors, Enums, and Memory Management

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 do not need to pass the array length as an additional argument.

Example:

```
public class ArrayLengthExample {

   // Method that prints the length of the array
   static void printLength(int[] arr) {
      System.out.println("Length: " + arr.length);
   }

   public static void main(String[] args) {
      int[] data = {10, 20, 30};
      printLength(data);
   }
}
```

Output: Length: 3

Passing Arrays to Methods (cont.)

- 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.

```
public class ChangeArrayElement {

    // Method that changes the first element of a String array
    static void changeFirstElement(String[] arr) {
        arr[0] = "Updated";
    }

    public static void main(String[] args) {
        String[] texts = {"Hello", "World"};
        changeFirstElement(texts);
        System.out.println(texts[0]);
    }
}
```

Output: Updated

- 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.

```
public class PrimitivePassExample {

    // Method that attempts to change the value
    static void tryToChange(int value) {
        value = 99;
    }
    public static void main(String[] args) {
        int[] nums = {10, 20, 30};
        tryToChange(nums[0]);
        System.out.println(nums[0]); }
}
```

Output: 10

Example – Passing Array

```
// Fig. 7.13: PassArray.java
// Passing arrays and individual array elements to methods.

public class PassArray {
   // main creates array and calls modifyArray and modifyElement
   public static void main(String[] args) {
      int[] array = {1, 2, 3, 4, 5};

      System.out.printf(
            "Effects of passing reference to entire array:%n" +
                  "The values of the original array are:%n");

      // output original array elements
      for (int value : array) {
         System.out.printf("   %d", value);
      }
```

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

Example – Passing Array

```
        modifyArray(array); // pass array reference
        System.out.printf("%n%nThe values of the modified array are:%n");
        // output modified array elements
        for (int value : array) {
           System.out.printf("   %d", value);
        }
        System.out.printf(
              "%n%nEffects of passing array element value:%n" +
                    "array[3] before modifyElement: %d%n", array[3]);

        modifyElement(array[3]); // attempt to modify array[3]
        System.out.printf(
              "array[3] after modifyElement: %d%n", array[3]);
     }
     public static void modifyArray(int[] array2) {
        for (int counter = 0; counter < array2.length; counter++) {
           array2[counter] = array2[counter]*2;
        }
     }
     public static void modifyElement(int element) {
        element *=2;
        System.out.printf( "Value of element in modify element: %d%n", element);
     }
}
```

```
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.)

7.11 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.

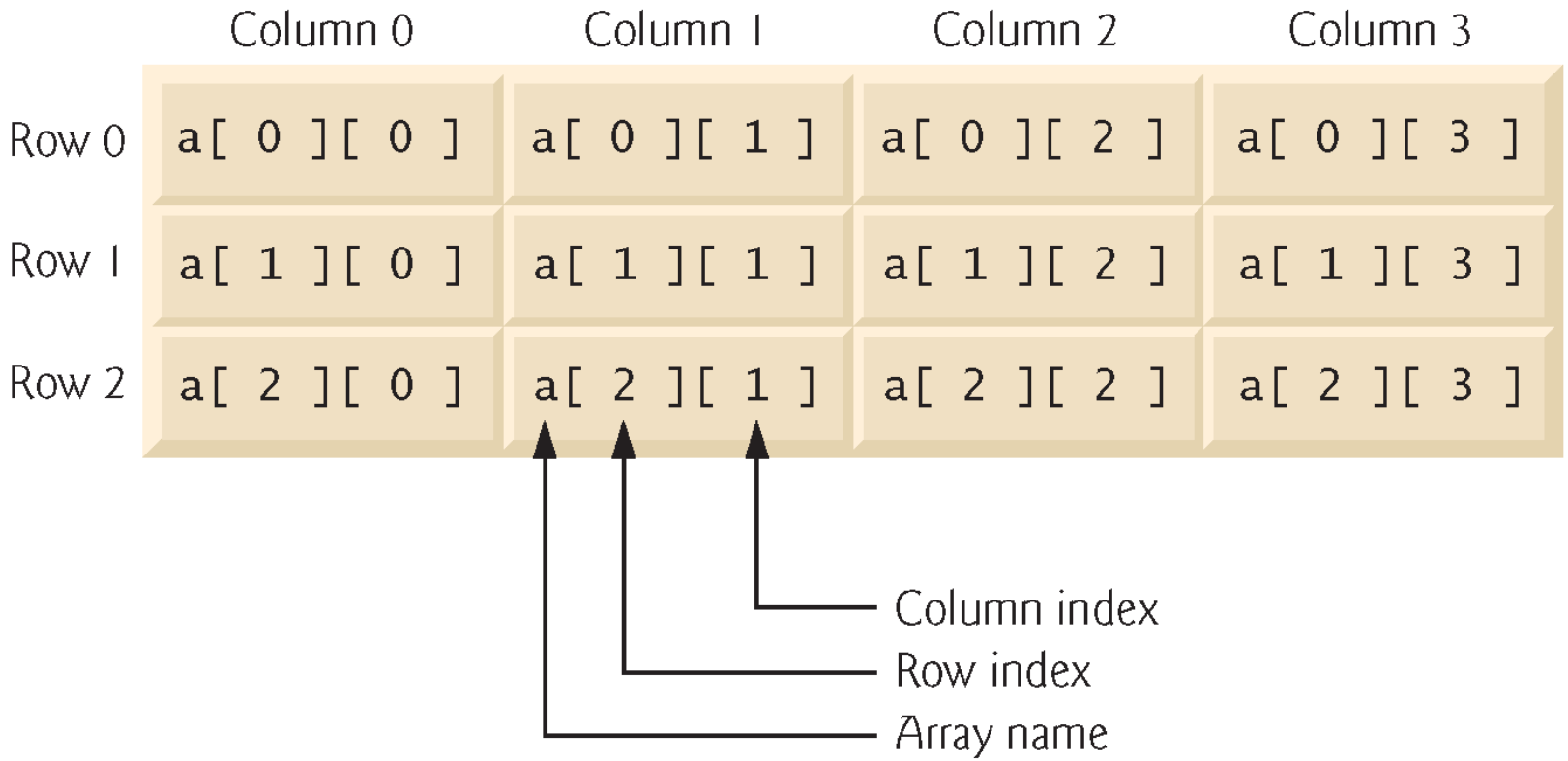


Fig. 7.16 | Two-dimensional array with three rows and four columns.

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 starting values are grouped by rows using curly brackets.
  - 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.*

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`).

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.

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.)

8.5 Overloaded Constructors

- Overloaded constructors enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide multiple constructor declarations with different signatures.
- Recall that the compiler differentiates signatures by the *number* of parameters, the *types* of the parameters and the *order* of the parameter types in each signature.
- The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration.

Example - Overloaded Constructors

```
public class Person {
    private String name;
    private int age;
    private String address;

    // Constructor with one parameter (name)
    public Person(String name) {
        this.name = name;
        this.age = 0; // Default age
        this.address = "Unknown"; // Default address
    }

    // Constructor with two parameters (name and age)
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        this.address = "Unknown"; // Default address
    }

    // Constructor with three parameters (name, age, and address)
    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Address: " + address);
    }
    public static void main(String[] args) {
        Person person1 = new Person("John");
        Person person2 = new Person("Alice", 25);
        Person person3 = new Person("Bob", 30, "New York");

        person1.displayInfo();
        person2.displayInfo();
        person3.displayInfo();
    }
}
```

Output:

```
Name: John
Age: 0
Address: Unknown
Name: Alice
Age: 25
Address: Unknown
Name: Bob
Age: 30
Address: New York
```

Overloaded Constructors (Cont.)

- A program can declare a so-called no-argument constructor that is invoked without arguments.
- Such a constructor simply initializes the object as specified in the constructor's body.
- Using `this` in method-call syntax as the first statement in a constructor's body invokes another constructor of the same class.
  - Popular way to *reuse* initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body.
- Once you declare any constructors in a class, the compiler will not provide a default constructor.

Example 2- Overloaded Constructors

```
public class Book {
    private String title;
    private int pages;

    // Constructor with parameters
    public Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    // No-arg constructor that calls another constructor
    public Book() {
        this("Unknown", 100); // reuses logic from the parameterized constructor
    }

    public void printInfo() {
        System.out.println("Title: " + title + ", Pages: " + pages);
    }

    public static void main(String[] args) {
        Book defaultBook = new Book();       // uses no-arg constructor
        Book specificBook = new Book("1984", 328); // uses param constructor

        defaultBook.printInfo();
        specificBook.printInfo();
    }
}
```

Output:
Title: Unknown, Pages: 100
Title: 1984, Pages: 328

Common Programming Error 8.3

A compilation error occurs if a program attempts to initialize an object of a class by passing the wrong number or types of arguments to the class's constructor.

8.9 enum Types

- **The basic enum type defines a set of constants represented as unique identifiers.**
- Like classes, all enum types are **reference** types.
- An enum type is declared with an enum declaration, which is a comma-separated list of *enum constants*

Enum Types (Cont.)

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are *implicitly* `final`.
  - `enum` constants are implicitly `static`.
  - Any attempt to create an object of an enum type with operator `new` results in a compilation error.
- `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements.

Enum Types (Cont.)

- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation.

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
public class EnumExample {
   public static void main(String[] args) {
      Day day = Day.MONDAY;

      // Convert enum constant to String
      String dayString = day.toString();

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

Output: MONDAY

Enum Types (Cont.)

For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants.

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}

public class EnumLoopExample {
    public static void main(String[] args) {
        for (Day day : Day.values()) {
            System.out.println(day);
        }
    }
}
```

Output:

```
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
```

Example: Enum Types

```
// Define an Enum for Access Levels
enum AccessLevel {
    ADMIN, MODERATOR, USER, GUEST
}

public class EnumExample {
    public static void main(String[] args) {
        // Assigning an enum constant to a variable
        AccessLevel userAccess = AccessLevel.MODERATOR;

        // Using if-else statement with enum
        if (userAccess == AccessLevel.ADMIN) {
            System.out.println("Full access granted.");
        } else if (userAccess == AccessLevel.MODERATOR) {
            System.out.println("Moderation access granted.");
        } else if (userAccess == AccessLevel.USER) {
            System.out.println("User access granted.");
        } else {
            System.out.println("Guest access granted.");
        }

        // Display all available access levels
        System.out.println("\nAvailable Access Levels:");
        for (AccessLevel level : AccessLevel.values()) {
            System.out.println(level);
        }
    }
}
```

Output:

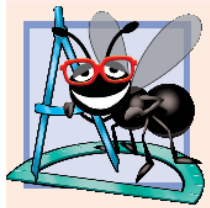
```
Moderation access granted.

Available Access Levels:
ADMIN
MODERATOR
USER
GUEST
```

8.10 Garbage Collection

- Every object uses system resources, such as memory.
  - Need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur.
- The JVM performs automatic garbage collection to reclaim the *memory* occupied by objects that are no longer used.
  - When there are *no more references* to an object, the object is *eligible* to be collected.
  - Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates.

Garbage Collection (Cont.)

- So, memory leaks that are common in other languages like C and C++ (because memory is *not* automatically reclaimed in those languages) are *less* likely in Java, but some can still happen in subtle ways.
- Resource leaks other than memory leaks can also occur.
  - If an app opens a file to modify it, the file stays "locked" until the app closes it.
  - If the app doesn't close the file properly, other apps can't access it until the original app finishes and releases it.



Software Engineering Observation 8.8

Many Java API classes (e.g., class `Scanner` and classes that read files from or write files to disk) provide `close` or `dispose` methods that programmers can call to release resources when they're no longer needed in a program.

Questions?