Chapter 8: Arrays

8.1 Array Elements

An **array** is an object that stores an ordered list of values of the same type in a single variable, with each value accessible through a numeric index.

Each element has a specific numbered position (index)

Arrays are zero-indexed in Java

- First element is at index 0
- Last element is at index (N-1) where N is array size

To access a value in an array, use the syntax: `arrayName[index]`

ex. `height[8]` refers to the 9th element

Array elements are stored contiguously (next to each other) in memory, which makes accessing them extremely efficient.

8.2 Declaring and Using Arrays

Consistent with other java objects, arrays must be instantiated using the `new` operator

```
int[] height = new int[11];
```

- Arrays hold elements of the same type, marked by `[]` after the type.
- Array size is fixed at creation and specified in initialization square brackets

Arrays can also be declared with the syntax `int grades[];` but this is rarely used.

Array Access & Bounds

Important concepts for array access:

- Square bracket operator [] has highest operator precedence
- Java performs automatic bounds checking
  - Valid indexes: 0 to (length-1)
  - Out of bounds access throws ArrayIndexOutOfBoundsException
- Array length accessible via .length constant

Array Initialization

Two ways to initialize arrays:

- Using new operator: `int[] array = new int[size];`
- Using initializer list: `int[] scores = {87, 98, 69};`
    - Must be used at declaration
    - Size determined by number of elements
- Even if an array is declared `final`, the contents of an array are still mutable.

Arrays as Parameters

An entire array can be passed as a parameter to a method.

Array parameters are passed by reference, meaning the method receive an alias of the reference

- Methods can modify array elements, which affects the original array.
- Individual elements follow normal parameter passing rules
    - Primitives passed by value
    - Objects passed by reference

8.3 Arrays of Objects

Arrays can store references to objects as elements. This enables complex data structures by combining arrays and objects.

e.g. `String[] words = new String[5];`

When declaring an array of objects, the initialization only creates *space* for object references. It does **not** create the actual objects. Initially all elements contain `null` references.

Each object stored in an array must be instantiated separately. eg:

```
//strings initialized with string literals.
String[] verbs = {"play", "work", "eat", "sleep"};

//strings initialized with `new` initialization statements
MyClass[] myArray = {
   new MyClass(1),
   new MyClass(2),
   new MyClass(3)
};
```

Alternatively, a method could be used to populate an array from an external data source such as a text file after it is initialized:

```
// Example of populating an array from a file
public class ArrayFileExample {
    public static String[] loadWordsFromFile(String filename) {
        try {
            Scanner scanner = new Scanner(new File(filename));
            ArrayList<String> wordList = new ArrayList<>();

            // Read file word by word
            while (scanner.hasNext()) {
                wordList.add(scanner.next().trim());
            }

            // Convert ArrayList to array
            String[] words = new String[wordList.size()];
            words = wordList.toArray(words);

            scanner.close();
            return words;

        } catch (FileNotFoundException e) {
            System.out.println("Error reading file: " + e.getMessage());
            return new String[0];
        }
    }

    public static void main(String[] args) {
        String[] vocabulary = loadWordsFromFile("words.txt");

        // Print loaded words
        for (String word : vocabulary) {
            System.out.println(word);    }
    }
}
```

8.4 Command-Line Arguments

Command-line arguments are values passed to a Java program through the `main` method's String[] parameter (typically called args).

The main method always receives a String[] parameter for command-line arguments

- Arguments are provided when running the program:
  - Command line input example: `java ProgramName arg1 arg2`
- Accessing Arguments:
  - Use array indexing: `args[0]`, `args[1]`, etc.
  - Arguments are always stored as Strings

Important Considerations:

- Error Handling:
  - ArrayIndexOutOfBoundsException occurs if accessing missing arguments
  - Extra arguments are stored but can be ignored
- Type Conversion:
  - Numeric input needs explicit conversion from String
- IDE Support:
  - Some development environments may handle command-line arguments differently
  - Consult IDE documentation for specific implementation

Example Usage:

```
public class NameTag {
  public static void main(String[] args) {
    System.out.println("     " + args[0]);
    System.out.println("My name is " + args[1]);
  }
}
```

Sample Output:

```
> java NameTag Hello Bill
    Hello
My name is Bill
```

API JAVA REFERENCE:

Here's a clear and concise table illustrating the differences between an **Array** and an **ArrayList** in Java, including a description, typical uses, and 5 common methods used in ArrayList along with their equivalent alternatives in Arrays:

Aspect	Array	ArrayList
Description	Fixed-size, static data structure used to store elements of a single data type.	Dynamic, resizable data structure that can grow or shrink automatically.
Type	Static (size is fixed at declaration)	Dynamic (size adjusts automatically)
Performance	Better performance for static, fixed-size data	Slightly slower due to resizing overhead; more flexible
Storage Type	Can store primitives and objects	Can store only object types (no primitives directly)
Uses	- When the number of elements is known upfront and won't change. - Performance-critical situations. - Primitive data handling.	- When the number of elements can change. - Frequent insertions or deletions. - Easier manipulation of data.

Common Methods and Alternatives:

Operation	ArrayList Method	Array Alternative
Add element	`list.add("Element")`	Manually assign at index: `array[index] = "Element";` (index must be known, cannot dynamically expand array)
Access/Get element	`list.get(index)`	Direct access by index: `array[index]`
Update element	`list.set(index, "Value")`	Direct assignment: `array[index] = "Value";`
Remove element	`list.remove(index)`	No direct removal (requires manual shifting or creating a new array) (e.g., manually shift elements or use arrays utility methods and copying)
Size/Length	`list.size()`	Use property: `array.length`