In previous chapters, we used single variables (like `int score` or `String name`) to hold data. This works well for individual pieces of information. However, complex programs often require managing large collections of related data, such as a list of 100 test scores or the names of 50 students. To handle these data structures efficiently, Java introduces the concept of **arrays**.

Arrays allow us to organize numerous data elements under a single variable name, enabling powerful processing capabilities, often utilizing loops (which you mastered in Chapter 4).

What you will learn

In this chapter, you will learn how to:

- Understand that arrays are **objects** that store a **fixed-size sequence of elements** of the same data type.
- Declare, create, and initialize arrays using default or custom values.
- Access and manipulate array elements using **subscripts** and understand **zero-based indexing**.
- Process array elements efficiently using `for` loops and the **enhanced `for` loop**.
- Understand the importance of **array bounds checking** and the `length` field.
- Pass arrays to methods and understand Java's **pass-by-reference** mechanism for objects.
- Work with arrays of objects, including `String` arrays and custom classes.
- Implement multi-dimensional structures using **two-dimensional arrays**.
- Understand the concept of **command-line arguments** passed to the `main` method. # 7.1 Introduction to Arrays

Primitive variables are designed to hold only one value at a time. **Arrays** allow you to create a collection of multiple values, provided those values are all of the same data type.

An array is a list of data elements. **An array is a Java object**. This means that when you declare an array variable, you are actually creating an object reference variable that stores a memory address, not the value itself.

There are three key facts about standard Java arrays: 1. **Homogeneous:** An array can store any type of data (primitives or objects) but only **one type of data** at a time. 2. **Fixed Size:** Once created, **an array size is fixed and cannot be changed**. You cannot add or remove items after initialization. 3. **Indexed:** Elements are accessed using an index position called a **subscript**, and **array indexes always start at 0**.

Declaring and Creating an Array

To use an array, you must first declare an **array reference variable** and then create the array object itself using the `new` keyword.

Syntax for Declaration and Creation:

```
dataType[] arrayName; // Declaration of the reference variable
arrayName = new dataType[size]; // Creation and assignment
```

The **array's size** indicates the number of elements the array can hold. The size must be a non-negative number, which can be a literal value, a constant, or a variable.

```
// Declaration of an array reference for integers
int[] scores;

// Creation: Allocate space for 10 integers (indexes 0 through 9)
scores = new int[10];
```

You can also declare an array reference and create it in the same statement:

```
int[] numbers = new int[6]; // Creates an array of 6 integers
```

For example, an array of six elements of type `int` will have the indexes 0, 1, 2, 3, 4, and 5.

Example: Top Artists

Imagine an application to track a list of **Top 4 music artists** of the month, which will be stored as an array of `String` objects.

```
public class TopArtistsChart {
    public static void main(String[] args) {
        // 1. Declaration and Creation in one step
        // We need 4 spots for 4 artists (indices 0, 1, 2, 3)
        String[] artistNames = new String[4];

        // 2. Default initialization (String arrays default to null)
        System.out.println("Default artist in the first spot (index 0): " + artistNames[0])
    }
}
```

If you initialize an array using the `new` keyword without specifying custom values, the elements are created using the **default values** of the data type:

Data Type	**Default Value**	**Example**
`int`, `byte`, `short`, `long`	**0**	`int[] counts = new int[2];` → `{0, 0}`

Data Type	Default Value	Example
`double`, `float`	**0.0**	`double[] sales = new double[2];` → `{0.0, 0.0}`
`boolean`	**false**	`boolean[] isTouring = new boolean[2];` → `{false, false}`
All Object Types (`String`, `Integer`, etc.)	**null**	`String[] names = new String[2];` → `{null, null}`

Initializing Arrays with Custom Values

You can initialize an array with custom values by specifying them in **curly brackets** when you declare the array:

```
dataType[] arrayName = {value1, value2, value3, ...};
```

When using this syntax, you **do not** use the `new` keyword or specify the size, as the compiler automatically determines the size based on the number of values provided.

Example: The Periodic Table's First Elements

We can initialize an array of strings to represent the first four elements of the periodic table:

```
public class ElementInitializer {
    public static void main(String[] args) {
        // Creates a String array of size 4 (index 0 through 3)
        String[] elements = {"Hydrogen", "Helium", "Lithium", "Beryllium"};

        // Accessing array elements using their subscripts (index positions)
        // Array elements are accessed by the array name and an index position within squar
        System.out.println("The element at index 0 is: " + elements[0]); // Hydrogen
        System.out.println("The element at index 2 is: " + elements[2]); // Lithium

        // Array elements can be treated as any other variable.
        String firstElement = elements[0];
        System.out.println("The first element is: " + firstElement);
    }
}
```

Check Your Understanding (7.1)

1. What are the three key properties that define a standard array in Java?
2. If you declare an array using `String[] names = new String[10];`, what is the default value stored in `names`?

3. How do you declare an array reference and create an array of 12 doubles in a single statement?
4. What is the subscript (index) of the *first* element and the *last* element in an array initialized as `char[] vowels = {'a', 'e', 'i', 'o', 'u'};`?

Practice Problems (7.1)

1. **Declaration:** Write a single Java statement to declare and initialize an array of integers named `highScores` with the values 98, 95, 99, 100, and 92.
2. **Access:** Write a statement that retrieves the value 95 from the `highScores` array created in the previous problem and stores it in an integer variable named `secondPlace`.
3. **Fixed Size:** Explain why the following code snippet would cause an error if run, focusing on the nature of array sizing:

```
int[] numbers = new int;
numbers = 4; // Attempting to assign to the 4th index
```

7.2 Accessing and Processing Array Elements

Once an array is created, its elements can be accessed, read, and modified. Array elements are referenced using their **subscript** (index). You can assign or re-assign values in the array using the index syntax.

The Array `length` Field

Arrays, as objects, provide a public **field** named `length`. This field contains the number of elements in the array.

> **Important Distinction:** Array length is a **field**, accessed without parentheses (`array.length`). This differs from the `String` class, which uses a **method** called `length()` to retrieve the number of characters.

Looping through Array Elements

Because the size of a standard array is known (fixed), looping through array elements is an ideal use case for a **`for` loop**.

The standard loop structure for processing every element in an array is:

```
for (int i = 0; i < array.length; i++) {
    // Access element at position i: arrayName[i]
}
```

By starting the counter `i` at 0 and using the condition `i < array.length`, you ensure the loop runs exactly `array.length` times, covering indexes `0` up to `array.length - 1`.

Example: Calculating the Score Total

Let's use a `for` loop and the `length` field to calculate the sum of test scores:

```java
public class ScoreAccumulator {
    public static void main(String[] args) {
        int[] scores = {85, 92, 78, 95, 88};
        int total = 0; // The accumulator variable

        // Loop runs from i=0 up to (but not including) 5
        for (int i = 0; i < scores.length; i++) {
            // Add the current score element to the running total
            total += scores[i];
        }

        System.out.println("Total scores processed: " + scores.length);
        System.out.println("Accumulated total: " + total);
    }
}
```

Bounds Checking

Java automatically performs **array bounds checking**, ensuring that a statement cannot use a subscript that is outside the range of valid subscripts.

- The first element is always at index **0**.
- The last element is always at index **`array.length - 1`**.

If you try to access an array element that is at `array.length` position or greater, Java will trigger an **`ArrayIndexOutOfBoundsException`**. This is a common error to watch out for, often resulting from an **off-by-one error**.

Tip: Avoid Printing "Item #0" to User

When looping through data, remember that computers start counting at 0, but humans start at 1.

Example: The Restaurant Order Builder

To maintain correct array indexing internally while providing user-friendly output, use `(i + 1)` when printing messages to the user:

```java
public class OrderBuilder {
    public static void main(String[] args) {
        // We know we want 5 items for the lunch order
        String[] orderItems = new String[5];
        Scanner keyboard = new Scanner(System.in);
        for (int i = 0; i < orderItems.length; i++) {
            // The user sees item #1, #2, #3, etc.
            System.out.print("Enter item #" + (i + 1) + ": ");
            orderItems[i] = keyboard.nextLine();
        }

        System.out.println("\n--- Final Order ---");
        // Print the contents of the array using a read-only loop (next section)
        for (String item : orderItems) {
            System.out.println(item);
        }
    }
}
```

The Enhanced `for` Loop (Read-Only)

Java provides a simplified array processing syntax, often called the **enhanced `for` loop** or **for-each loop**. This syntax is useful for **reading elements only**, as it does not allow you to refer to elements by their index, which prevents re-assigning values.

Syntax:

```
for (dataType elementVariable : arrayName) {
    // Statements using elementVariable
}
```

Example: Printing a List of Musical Artists

We use the enhanced `for` loop to easily display the contents of a String array:

```
public class MusicArtists {
    public static void main(String[] args) {
        String[] artists = {"Steely Dan", "The Clash", "David Bowie"};

        System.out.println("--- Artists ---");
        for (String artist : artists) {
            System.out.println("Artist: " + artist);
        }
    }
}
```

User-specified Array Length and Elements

Although arrays are fixed in length once created, you can allow a user to determine the size and elements of an array during runtime using the `Scanner` object.

Example: Building a Recipe Array

This code prompts a user for the number of ingredients and dynamically creates the array size based on that input:

```java
import java.util.Scanner;

public class RecipeCreator {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("How many ingredients are in your recipe? ");
        int size = keyboard.nextInt();
        keyboard.nextLine(); // Consume newline

        // Dynamically create the array based on user input
        String[] ingredients = new String[size];

        for (int i = 0; i < ingredients.length; i++) {
            System.out.print("Enter ingredient " + (i + 1) + ": ");
            ingredients[i] = keyboard.nextLine();
        }

        System.out.println("\n--- Your Recipe ---");
        // Use enhanced loop to display
        for (String item : ingredients) {
            System.out.println(item);
        }
    }
}
```

Check Your Understanding (7.2)

1. What is the fundamental difference in syntax when accessing the length of a `String` variable versus the length of an array?
2. If `int[] data = new int;`, what is the valid range of index numbers? What is the index of the element accessed by the statement `data[data.length - 1]`?
3. Explain why the enhanced `for` loop cannot be used to initialize or change the values of an array's elements.
4. How would you prevent an `ArrayIndexOutOfBoundsException` when using a `for` loop to iterate through an array of unknown size?

Practice Problems (7.2)

1. **Loop Trace:** How many times will the `System.out.println` statement execute, and what is the final output of the following code?

```java
int[] values = {10, 20, 30, 40};
```

```java
for (int i = 0; i < 4; i++) {
    System.out.println(values[i]);
}
```

2. **Indexing:** Given `String[] planets = {"Mars", "Earth", "Venus"};`, write a single statement that reassigns the element "Earth" to "Terra" (the last element's index is always at `array.length - 1`).
3. **Bounds Checking:** Identify the statement that would cause an `ArrayIndexOutOfBoundsException` in the following code, assuming `cars` is an array of size 3:

```java
Car[] cars = new Car[3];
cars[0] = new Car();
cars[1] = new Car();
cars[2] = new Car();
cars[3] = new Car();
```

7.3 Passing and Returning Arrays to and from Methods

Passing Arrays to Methods

Arrays are objects in Java. When you pass an object as an argument to a method, Java follows the **pass-by-value** rule, but the value passed is actually a copy of the object's **memory address (reference)**.

When an array's **reference variable** is passed to a method, the method receives this memory address, giving it **direct access to the original array in memory**. Therefore, the receiving method can **change values in the original array**.

Critical Distinction: Array vs. Primitive Element

- **Passing the array reference:** The method receives a copy of the memory address and can modify the ***contents*** (elements) of the original array.
- **Passing an *individual element*** (if the element is a primitive type like `int` or `double`): The method receives only a copy of the ***value***, and changes to that value will **not** affect the original array.

Example: Student Test Scaling Scores

Imagine an application that records student test scores. A method is used to adjust these scores by applying a curve. The curve is calculated by finding the difference between the highest score and 100, and adding that difference to every score.

```java
public class TestScoreScaler {
    /**
```

```
     * Accepts an array of test scores and applies a curve.
     * The curve is calculated by finding the difference between the highest score and 100,
     * and adding that difference to every score.
     * @param scores The array of student test scores.
     */
    public static void applyBonus(int[] scores) {
        System.out.println("\n--- Analyzing Scores for Curve ---");

        // 1. Find the maximum score in the array
        int maxScore = 0;
        for (int score : scores) {
            if (score > maxScore) {
                maxScore = score;
            }
        }

        // 2. Calculate the scale factor (difference to 100)
        int scaleAmount = 100 - maxScore;
        System.out.println("Highest Score: " + maxScore);
        System.out.println("Scale Amount: +" + scaleAmount + " points");

        System.out.println("\n--- Applying Scale ---");
        // 3. Apply the scale to all scores
        for (int i = 0; i < scores.length; i++) {
            scores[i] += scaleAmount;
        }
        System.out.println("--- Scale Applied ---");
    }

    public static void main(String[] args) {
        // Scores for 4 students
        int[] testScores = {72, 88, 91, 65};

        System.out.print("Original Scores: ");
        for (int score : testScores) {
            System.out.print(score + " ");
        }

        // Pass the array reference to the method
        applyBonus(testScores);

        System.out.print("Scaled Scores:   ");
        // The original array has been permanently modified
        for (int score : testScores) {
            System.out.print(score + " ");
        }
        System.out.println();
    }
}
```

In the above code, the `main` method creates the array referenced by `testScores`. This variable holds the **memory address** (the reference) of the actual scores. If you were to print `testScores`, you would see something like `[I@372f7a8d` (a memory address).

When the statement **`applyBonus(testScores)`** is executed, the **value** of the reference in **`testScores`** is copied into the method parameter, **`scores`**.

- Both **`testScores`** and **`scores`** now point to the **same array data** in memory.

Inside the **`applyBonus`** method, the `for` loop uses the **`scores`** reference to directly access and modify the elements. Specifically, the line **`scores[i] += scaleAmount;`** modifies the actual array data shared by both references.

After **`applyBonus`** returns, the **`testScores`** variable in **`main`** still points to the *same* memory location, which now holds the modified values, resulting in the final output.

Check Your Understanding (7.3 - Passing Arrays)

1. When an array reference variable is passed to a method, what specific piece of information does the parameter variable hold?
2. If Method A calls Method B, passing an array reference, and Method B modifies the array contents, does this modification persist when control returns to Method A? Why or why not?
3. If you pass a single element of an array, such as `arr`, to a method, can that method change the original value stored at `arr` if `arr` holds primitive data types? Explain.

Practice Problems (7.3 - Passing Arrays)

1. **Code Trace:** Assume the method `zeroOut(int[] list)` sets every element in the passed array to 0. If you call `zeroOut(temperatures)` where `temperatures` is ‘`, what are the values of`temperatures‘ after the method call?
2. **Method Header:** Write the header for a static method named `printArrayElements` that accepts one argument: an array of `String` objects.

Returning Arrays from Methods

Just as methods can return primitive values or references to objects, they can also return references to arrays. To define such a method, you must specify the array's data type followed by empty square brackets (`[]`) as the return type.

Example: The Recipe Organizer Returns an Array

We can encapsulate the user input logic from earlier into a method that returns the finalized array of ingredients:

```java
import java.util.Scanner;

public class RecipeOrganizer {
    // The return type is String[]
    public static String[] getIngredients(int numIngredients) {
        Scanner keyboard = new Scanner(System.in);
        String[] ingredients = new String[numIngredients];

        for (int i = 0; i < ingredients.length; i++) {
            System.out.print("Enter ingredient " + (i + 1) + ": ");
            ingredients[i] = keyboard.nextLine();
        }
        return ingredients; // Return the reference to the newly created String array
    }

    public static void main(String[] args) {
        int recipeSize = 3;
        // The return value (the array reference) is assigned to the 'recipe' array
        String[] recipe = getIngredients(recipeSize);

        System.out.println("\n--- Finalized Recipe ---");
        // Display the returned array's contents
        for (String item : recipe) {
            System.out.println("-> " + item);
        }
    }
}
```

Check Your Understanding (7.3 - Returning Arrays)

1. When a method returns an array, is a copy of the entire array data returned, or is something else returned?
2. Write the method header for a static method named `getDailyTemperatures` that returns an array of `double` values.

Practice Problems (7.3 - Returning Arrays)

1. **Error Correction:** The following method intends to return an array of integers but contains a syntax error. Fix the code.

```java
public static int returnFive() {
    int[] fiveNumbers = {1, 2, 3, 4, 5};
    return fiveNumbers;
}
```

7.4 String Arrays and Arrays of Objects

Arrays can store references to Java objects, including the special `String` class and custom objects you create (Chapter 6).

String Arrays

A String array holds references to multiple `String` objects. Since each element of a String array is itself a `String` object, you can call any `String` method directly on that element using dot notation.

Example: Preparing an Artist List for Sorting

This example demonstrates how to iterate through a **String array**, use methods like `startsWith()` and `substring()` to clean the data (removing "The" for better sorting):

```
public class ArtistList {
    public static void main(String[] args) {
        // Initial list of artists
        String[] artists = {
            "Steely Dan", "The Clash", "The Misfits", "David Bowie",
            "The Smiths", "The Cure", "Jimi Hendrix", "New Order",
            "The Velvet Underground", "Queen", "Cocteau Twins"
        };

        System.out.println("--- Original List ---");
        for (String name : artists) {
            System.out.println(name);
        }

        System.out.println("\n--- Data Cleanup: Removing 'The ' ---");

        // Use a standard loop to process and clean up the data
        for (int i = 0; i < artists.length; i++) {
            String currentArtist = artists[i];
            // Check if the current artist name starts with "The " (case sensitive)
            if (currentArtist.startsWith("The ")) {

                // Get the substring starting at index 4 (right after "The ")
                String cleanedName = currentArtist.substring(4);

                // Reassign the array element with the cleaned string
                artists[i] = cleanedName;
                System.out.println("Changed: " + currentArtist + " -> " + cleanedName);
            }
        }

        System.out.println("\n--- Updated List ---");
        for (String name : artists) {
            System.out.println(name);
        }
    }
}
```

Arrays of Custom Objects

Arrays can also store references to custom objects, such as `Car` or `Song` objects. When you create an array of objects, you are only reserving space for the *references* (memory addresses); you must individually initialize each object within the array.

Example: The Car Lot Simulation

If we assume a fully coded `Car` class exists with a setter method `setAvailable(boolean status)`, we can create an array of `Car` objects:

```
// Assume Car class is defined elsewhere
public class CarLot {
    public static void main(String[] args) {
        // 1. Create an array reference to hold 5 Car objects (size 5)
        Car[] cars = new Car[5];

        // 2. Instantiate each Car object individually (required for objects!)
        cars[0] = new Car("Mustang", 2023);
        cars[1] = new Car("Civic", 2024);
        cars[2] = new Car("Model 3", 2022);
        cars[3] = new Car("F-150", 2021);
        cars[4] = new Car("Focus", 2023);

        // 3. Use an instance method on an element to modify its state
        // The index for the 4th car is 3 (index 0, 1, 2, 3)
        cars[3].setAvailable(false); // Mark the F-150 as sold

        // 4. Check the status of the 4th car (index 3)
        if (!cars[3].isAvailable()) {
            System.out.println("Car at index 3 (F-150) is marked unavailable.");
        }
    }
}
```

Check Your Understanding (7.4)

1. If `String[] words = {"alpha", "beta", "gamma"};`, what is the index of the element containing the string "beta"? Write the code to convert that element to all uppercase letters.
2. When you declare an array of custom objects, such as `Song[] playlist = new Song;`, why do you still need to use the `new` keyword ten times inside a loop or individually to fully initialize the array elements?

Practice Problems (7.4)

1. **Object Access:** Assume a `User` class has a public instance method `getUsername()`. Write the correct code snippet to retrieve and print the username of the user stored at index 0 of an array named `userList`.

```
User[] userList = new User[10];
userList[0] = new User("Neo");
// Write the print statement here
```

2. **String Method Check:** Given `String[] fruit = {"apple", "Banana", "Cherry"};`, write an `if` statement that checks if the string at index 1 is equal to "banana", ignoring case differences. # 7.5 Two-Dimensional Arrays

Arrays can be extended to model tabular data, like spreadsheets, grids, or matrices. These structures are known as **two-dimensional arrays** (or multidimensional arrays).

A two-dimensional array is essentially an array where each element is itself an array. Data is accessed using two subscripts: one for the **row** and one for the **column**.

Declaration and Initialization

Declaration Syntax:

```
dataType[][] arrayName;
```

Initialization Syntax:

You can initialize a 2D array by specifying the number of rows and columns:

```
int[][] matrix = new int[3][4]; // 3 rows, 4 columns
```

Or you can use initialization list syntax:

```
// 3 rows, 4 columns
int[][] scores = {
    {10, 20, 30, 40}, // Row 0
    {50, 60, 70, 80}, // Row 1
    {90, 100, 110, 120} // Row 2
};
```

Accessing Elements and Iterating

Elements are accessed using two index positions: `arrayName[row][column]`. Iterating over a 2D array requires the use of **nested `for` loops**:

- The **outer loop** typically iterates through the rows.
- The **inner loop** typically iterates through the columns within the current row.

Example: The Star Map Grid

This example initializes a 4x3 grid representing star map coordinates and uses nested loops to print every element:

```
public class StarMap {
    public static void main(String[] args) {
        // 4 rows (star clusters), 3 columns (x, y, z coordinates)
        double[][] coordinates = {
            {10.5, 20.0, 5.2},
            {1.1, 8.9, 15.0},
            {25.7, 3.3, 1.9},
            {40.0, 40.0, 40.0}
        };

        // Outer loop iterates through rows (using coordinates.length)
        for (int row = 0; row < coordinates.length; row++) {
            System.out.print("Cluster " + row + " coordinates: ");

            // Inner loop iterates through columns of the current row
            // coordinates[row].length gives the length of the inner array (columns)
            for (int col = 0; col < coordinates[row].length; col++) {
                System.out.printf("[%.1f] ", coordinates[row][col]); // Use printf for clea
            }
            System.out.println(); // Move to the next line after finishing a row
        }
    }
}
```

Ragged Arrays

Java allows you to create **ragged arrays**, where each row can have a different number of columns. This is achieved by initializing only the row count in the outer dimension, and then initializing each inner array individually:

```
int[][] raggedArray = new int[3][]; // 3 rows, but columns not yet defined
raggedArray[0] = new int[5]; // Row 0 has 5 columns
raggedArray[1] = new int[2]; // Row 1 has 2 columns
raggedArray[2] = new int[4]; // Row 2 has 4 columns
```

Ragged arrays are particularly useful when modeling structures that are not perfectly rectangular. For instance, a theater often has rows of varying lengths to accommodate the shape of the room (e.g., a curved front row might have fewer seats than the back row). Using a ragged array allows you to allocate exactly the memory needed for each row without wasting space on "empty" seats that don't exist.

```
// Modeling a theater with 3 rows of varying lengths
int[][] theaterSeats = new int[3][];

theaterSeats[0] = new int[10]; // Front row: 10 seats
theaterSeats[1] = new int[15]; // Middle row: 15 seats
theaterSeats[2] = new int[20]; // Back row: 20 seats
```

Check Your Understanding (7.5)

1. In the declaration `int[][] data = new int[5][8];`, what do the values 5 and 8 represent?
2. Why are nested loops required to effectively iterate over a two-dimensional array?
3. Define a ragged array in the context of Java.

Practice Problems (7.5)

1. **Access:** Given the array `char[][] grid = {{'A', 'B'}, {'C', 'D'}};`, write a statement to access and print the character 'D'.
2. **Declaration:** Write the statement to declare a two-dimensional array of `String` objects named `roster` with 10 rows and 3 columns. # 7.6 The ArrayList Class and the Arrays Utility Class

While standard arrays are powerful, they have limitations: their size is fixed upon creation, and they don't come with built-in methods for common tasks like sorting or printing. Java provides two helpful tools to address these needs: the `ArrayList` class and the `Arrays` utility class.

The ArrayList Class

The `ArrayList` class is part of the Java API (`java.util` package). Unlike a standard array, an `ArrayList` can **automatically resize** itself. You can add or remove elements, and the list will grow or shrink as needed.

Key Differences:

- **Dynamic Size:** Can grow and shrink.
- **Objects Only:** Can only store objects, not primitives (use Wrapper classes like `Integer` or `Double` for numbers).
- **Methods:** Uses methods like `.add()`, `.get()`, and `.size()` instead of square brackets and `.length`.

Example: A Dynamic Shopping List

```java
import java.util.ArrayList;

public class ShoppingList {
    public static void main(String[] args) {
        // Create an ArrayList to hold String objects
        ArrayList<String> list = new ArrayList<>();

        // Add items to the list
        list.add("Apples");
        list.add("Bread");
        list.add("Milk");

        System.out.println("List size: " + list.size()); // Output: 3

        // Access an item using .get(index)
        System.out.println("First item: " + list.get(0));

        // Remove an item
        list.remove("Bread");

        System.out.println("Updated List:");
        // Iterate using the enhanced for loop
        for (String item : list) {
            System.out.println("- " + item);
        }
    }
}
```

The Arrays Utility Class

The `java.util.Arrays` class provides a collection of static methods that perform common operations on standard arrays, such as sorting and searching.

Example: Sorting the Artist List

In an earlier section, we worked with a list of artists. Let's use the `Arrays.sort()` method to organize that list alphabetically, and `Arrays.toString()` to easily print the entire array without a loop.

```java
import java.util.Arrays;

public class ArtistSorter {
    public static void main(String[] args) {
        String[] artists = {
            "Steely Dan", "The Clash", "The Misfits", "David Bowie",
            "The Smiths", "The Cure", "Jimi Hendrix", "New Order",
            "The Velvet Underground", "Queen", "Cocteau Twins"
        };

        System.out.println("--- Before Sorting ---");
        // Arrays.toString() creates a nice string representation like [A, B, C]
        System.out.println(Arrays.toString(artists));

        // Sort the array in ascending order (A-Z)
        Arrays.sort(artists);

        System.out.println("\n--- After Sorting ---");
        System.out.println(Arrays.toString(artists));

        // We can also search for an artist using binarySearch (array must be sorted first!
        int index = Arrays.binarySearch(artists, "David Bowie");
        if (index >= 0) {
            System.out.println("\nFound 'David Bowie' at index: " + index);
        }
    }
}
```

Check Your Understanding (7.6)

1. What is the main advantage of an `ArrayList` over a standard array?
2. Why must you use `ArrayList<Integer>` instead of `ArrayList<int>`?
3. Which static method in the `Arrays` class allows you to rearrange elements into ascending order?

Practice Problems (7.6)

1. **ArrayList:** Write the code to create an `ArrayList` of `Double` values named `prices` and add the value `19.99` to it.
2. **Arrays Class:** Given an array `int[] numbers = {5, 1, 4, 2, 3};`, write the code to sort this array and then print it using `Arrays.toString()`.

7.7 Command-Line Arguments

After enduring months of instruction, you are now ready to learn the secret meaning of the method header that begins every Java executable class: `public static void main(String[] args)`.

The `String[] args` parameter means that the `main` method is configured to receive an array of `String` objects, referred to as `args`. These are the **command-line arguments** passed to your program when it is executed outside of an IDE.

This allows users to provide input to the program immediately upon execution.

You access these arguments positionally, just like any other array, using the index: `args`, `args`, etc..

Warning: If a user runs your program without providing the expected arguments, attempting to access a missing index (e.g., `args` when no arguments were provided) will result in an **`ArrayIndexOutOfBoundsException`**.

Example: The Program Launcher

If you compile and run the following program from the command line:

```java
public class ProgramLauncher {
    // The main method expects an array of strings called args
    public static void main(String[] args) {
        // Check if at least one argument was provided
        if (args.length > 0) {
            String name = args[0]; // Access the first argument

            // Access the length of the args array field
            System.out.println("Hello, " + name + "!");
            System.out.println("You supplied " + args.length + " command line arguments.");
        }
        else {
            System.out.println("Please run this program with your name as an argument.");
            System.out.println("Example: java ProgramLauncher Sarah");
        }
    }
}
```

Running the Program (Command Line):

Command	Output
`java ProgramLauncher Max`	Hello, Max! You supplied 1 command line arguments.
`java ProgramLauncher Data File 2024`	Hello, Data¡ You supplied 3 command line arguments.

Check Your Understanding (7.7)

1. What does the `String[] args` portion of the `main` method header represent?
2. If a user executes a program but provides no command-line arguments, what value is stored in `args.length`?

Practice Problems (7.7)

1. **Conversion:** If a user passes the string “50” as the first command-line argument, write the code snippet required to convert this string argument into an integer variable named `startValue`.
2. **Access:** Write a check that ensures at least three command-line arguments have been provided before attempting to access `args`.