GUVI | HCL
Skill Up. Level Up

# Arrays ,String API & Wrapper Classes

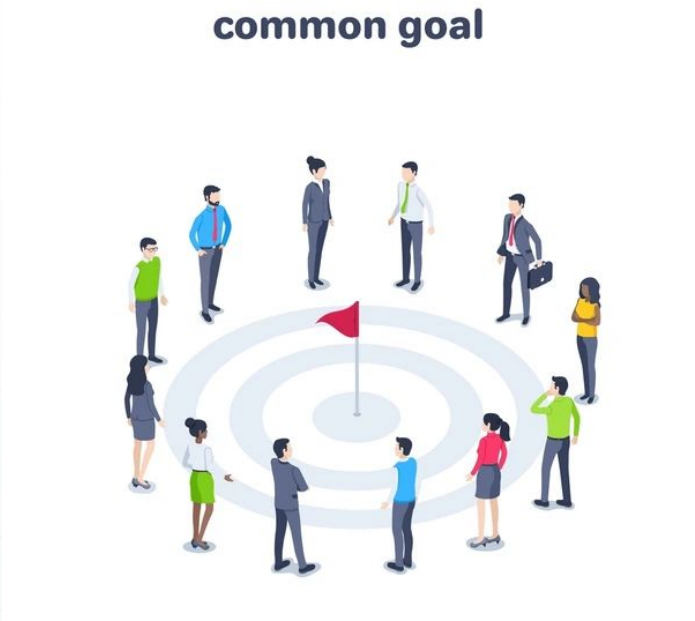
Programming Foundation

Presented by
**Nikhil Nair**

Website
**www.guvi.com**

# Objectives

- What you will learn today
  - Arrays & an introduction to 2D Arrays
  - Core String operations and the idea of immutability
  - Wrapper classes and the basics of autoboxing and unboxing
- How this builds on Lecture 2
  - Reuse loops and our “function” habits to process arrays and strings
  - We focus on small, single-purpose methods that return values



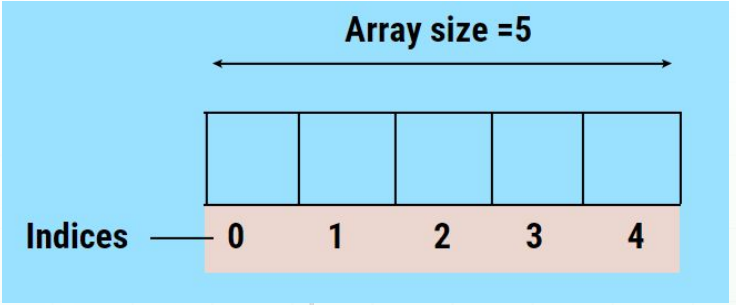
# Functions to Methods

- Let’s align the terms
    - A function is a self-contained block of code designed to perform a specific task
    - The relevant concept in Java is that of a method, because all methods must belong to a class
- Two ways methods show up
    - Static methods you can call without creating an object
    - Instance methods you call on an object you already have (we’ll cover this later)
- What a method’s signature tells you
    - Name and parameter list describe the inputs
    - Return type describes the output
    - Recall: a good method does one clear job and returns a value instead of printing

# Arrays

# Arrays: The Mental Model

- What is an array?
  - A fixed-size, ordered collection of elements of the same type
  - Addressed by 0-based indices
- What does not change after creation?
  - The length is fixed once you allocate the array
  - Reads and writes by index are constant time
- How arrays live in memory
  - Arrays are objects, so a variable holds a reference to the array
  - Valid indices are 0 through array length - 1; anything else will give an error
- When arrays fit well
  - The number of items is known or stable
  - You need predictable indexed traversal or in-place updates




```
int[] a = {10, 20, 30};
System.out.println(a[0]);      // 10
System.out.println(a.length);  // 3
```

# Declaring and Creating Arrays

- Declaration names a variable whose type is “array of T.”
    - Common form for readability is int[] a.
    - int a[] is also valid but less consistent with multi-variable declarations.
- Creation allocates the array object with a chosen length.
    - Use new int[n] when you know the size at runtime
    - Use a literal like {10, 20, 30} when you know the values upfront
- The length is fixed after creation
    - Plan the size or allocate a new array if the size must change
- Variables hold references to array objects
    - Assigning one array variable to another copies the reference, not the elements

```
public class Demo {
  public static void main(String[] args) {
    int[] a = new int[3];         // defaults: 0, 0, 0
    System.out.println(a.length); // 3
    int[] b = a;                  // same underlying array
    b[1] = 42;
    System.out.println(a[1]);     // 42
  }
}
```

# Indexing and Length

- Valid indices run from 0 to length - 1
    - Access outside this range is an error during program execution
- length is a field on arrays
    - Read it as arr.length without parentheses
    - Contrast with String.length() which is a method
- Check edge cases explicitly
    - An empty array has length == 0 and no last index to read
- Reading length is constant time
    - Prefer i < arr.length rather than hard-coding bounds

```
public class Demo {
  public static void main(String[] args) {
    int[] a = {10, 20, 30, 40};
    if (a.length > 0) {
      int last = a[a.length - 1];
      System.out.println(last);   // 40
    }
  }
}
```

# Iterating Arrays: Indexed For

- Use an indexed for loop when you need positions or plan to update elements
    - Typical pattern starts at 0, continues while < length, and increments by 1
- Boundaries must be precise to avoid off-by-one errors
    - Prefer i < arr.length as the end condition
- Indexed loops are flexible
    - Reverse iteration from length - 1 down to 0
    - Partial ranges or steps greater than one when the task demands it
- Control flow tools from Lecture 2 still apply
    - Use break to stop early and continue to skip specific positions when it improves clarity

```
public class Demo {
  public static void main(String[] args) {
    int[] a = {2, 4, 6};
    for (int i = 0; i < a.length; i++) {
      a[i] = a[i] * 2;          // in-place update
    }
    System.out.println(a[0] + " " + a[1] + " " + a[2]); // 4 8 12
  }
}
```

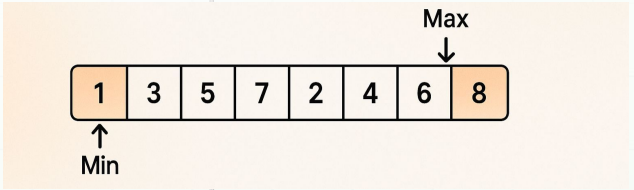
# Array Algorithms: Linear Search

- Purpose and behavior
  - Return the index of the first element equal to a given target
  - If the target is not found, return -1 as a clear sentinel
- Match rule
  - For primitives like int, compare with ==
  - For Strings, compare content with equals
- Steps to follow
  - Loop from index 0 while < array.length and check each element
  - Return the index immediately when you find a match
  - After the loop ends with no match, return -1
- Edge behavior
  - Empty arrays return -1 because nothing can match
  - When there are multiple matches, return the first index

```java
public class LinearSearchDemo {
  static int indexOf(int[] arr, int target) {
    if (arr == null) return -1;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] == target) return i;
    }
    return -1;
  }
  public static void main(String[] args) {
    System.out.println(indexOf(new int[]{4, 7, 7}, 7)); // 1
    System.out.println(indexOf(new int[]{4, 7, 7}, 5)); // -1
  }
}
```

# Activity: Max and Min

- What you will build
  - A method to return the largest value in an int[]
  - A method to return the smallest value in an int[]
- Hints for a correct first pass
  - Start from the first element and track a “best so far”
  - Update the best when you see a better candidate
- Edge rules we will use today
  - If the array is empty or null, return a documented sentinel
  - For max use Integer.MIN_VALUE; for min use Integer.MAX_VALUE



# Activity: Average

- What you will build
    - A method that returns the average of all elements in an int[]
- Hints for accurate results
    - Add up everything first, then divide once at the end
    - Use a wider running total to reduce overflow risk
- Edge rules we will use today
    - If the array is empty, return 0.0 as “no data yet”
    - Make the return type double to keep fractional parts

# 2D Arrays

# 2D Arrays: Arrays of Arrays

- What's a 2D array?
  - An array whose elements are arrays - think a matrix
  - Access items with [row][col]
- Shapes you may see
  - Rectangular: every row has the same length
  - Jagged: rows can have different lengths
- Lengths to remember
  - arr.length is the number of rows
  - arr[row].length is the number of columns in that row
- Practical notes
  - Each row is a separate array object
  - Default values apply just like in 1D arrays

```
public class MatrixIntro {
  public static void main(String[] args) {
    int[][] m = { {1, 2}, {3, 4, 5} };   // jagged
    System.out.println(m.length);        // rows: 2
    System.out.println(m[0].length);     // cols in row 0: 2
    System.out.println(m[1].length);     // cols in row 1: 3
    System.out.println(m[1][2]);         // 5
  }
}
```

# 2D Traversal Patterns

- Row-major traversal
  - Outer loop over rows from 0 to < arr.length
  - Inner loop over columns from 0 to < arr[row].length
- Jagged-safe bounds
  - Read the current row's length inside the inner loop
  - Guard against empty arrays and rows with length 0
- Common tasks you can do
  - Sum or count values across the grid
  - Search for a target and report its [row, col]
  - Print the grid in a readable form

```java
public class MatrixTraverse {
  public static void main(String[] args) {
    int[][] m = { {1, 2}, {3, 4, 5} };
    for (int r = 0; r < m.length; r++) {
      for (int c = 0; c < m[r].length; c++) {
        System.out.print(m[r][c] + " ");
      }
      System.out.println();
    }
  }
}
```

# Strings

# What are Strings?

- A String is an object that represents a sequence of characters
- It is a reference type, not a primitive
- Strings & Immutability
    - A String never changes after it is created
    - Any apparent change creates a new String with the new content
- Why this matters in your code
    - Repeatedly creating Strings in a loop can create several temporary Strings
    - Passing a String to a method cannot modify the original text
- Practical rules for this course
    - Use String for fixed or small pieces of text
    - Prefer a builder when assembling text in loops or many steps

```
public class App {
  public static void main(String[] args) {
    String s = "hi";
    String t = s + "!";
    System.out.println(s); // hi
    System.out.println(t); // hi!
  }
}
```

# Core String Methods

- Measuring and indexing
    - length() returns the number of characters
    - charAt(i) returns the character at index i where 0 ≤ i < length()
- Taking slices
    - substring(start, end) returns text from start up to but not including end
    - substring(start) returns from start to the end
- Searching
    - indexOf(x) returns the first position of x or -1 if not found
    - You can search for a single character or a substring

```
public class App {
  public static void main(String[] args) {
    String s = "hello";
    System.out.println(s.length());        // 5
    System.out.println(s.charAt(1));       // e
    System.out.println(s.substring(1, 4)); // ell
    System.out.println(s.indexOf("lo"));   // 3
  }
}
```

# Equality: == vs equals

- Two different questions
  - == asks whether two references point to the same object
  - equals asks whether two Strings have the same sequence of characters
- Safe rule for Strings
  - Use equals for content checks
  - Use equalsIgnoreCase only when case does not matter
- Pitfalls to avoid
  - name == "Admin" may appear to work sometimes, then fail later
  - If a variable might be null, write "Admin".equals(name) to avoid errors

```
public class App {
  public static void main(String[] args) {
    String a = new String("Admin");
    String b = "Admin";
    System.out.println(a == b);          // false (usually)
    System.out.println(a.equals(b));     // true
    String name = null;
    System.out.println("Admin".equals(name)); // false,
safe
  }
}
```

# StringBuilder vs StringBuffer

- What are they?
  - StringBuilder and StringBuffer are mutable helpers to build text piece by piece
  - They reduce temporary String objects created by repeated + in loops
- Which one to use?
  - Prefer StringBuilder for single-threaded, everyday code
  - StringBuffer is the older, thread-safe variant; we won't be using it
- How to use them safely
  - Append parts as you go, then call toString() once at the end to get a String
  - Use a builder when joining many small parts; use + for one-off, small concatenations

```
public class StringBuilderDemo {
  public static void main(String[] args) {
    String[] parts = {"a", "b", "c"};
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < parts.length; i++) {
      if (i > 0) sb.append(",");
      sb.append(parts[i]);
    }
    String out = sb.toString();
    System.out.println(out); // a,b,c
  }
}
```

# Wrapper Classes

# Wrapper Classes: Why and What

- What is a wrapper?
    - An object version of a primitive, for example Integer, Double, Character, Boolean
    - It holds a primitive value inside an object
- Why wrappers exist?
    - Some APIs expect objects rather than primitives
    - Wrappers can be null to mean “no value”, primitives cannot be null
- Habits and trade-offs
    - Prefer primitives for plain calculations and flags
    - Use wrappers only when an API or design truly needs an object
    - Compare wrapper values with equals, not ==

```
public class WrapperIntroDemo {
  public static void main(String[] args) {
    int p = 5;                  // primitive
    Integer w = Integer.valueOf(5); // wrapper

System.out.println(w.equals(Integer.valueOf(p))); //
true
  }
}
```

# Autoboxing and Unboxing

- What do these mean?
    - Autoboxing: Java automatically converts a primitive to its wrapper when an object is expected
    - Unboxing: Java automatically converts a wrapper to its primitive when a primitive is expected
- Where you will see this
    - Assigning int into an Integer variable or passing it to a method that expects Integer
    - Using an Integer where an int is needed triggers unboxing
- Rules to keep code safe
    - Never unbox a null wrapper; check for null and supply a default value
    - Use equals for comparing wrapper values; use == for primitives
    - Avoid unnecessary boxing in tight loops or simple arithmetic

```
public class BoxingDemo {
  static int safeUnbox(Integer maybe) {
    return (maybe != null) ? maybe : 0; // default
when null
  }
  public static void main(String[] args) {
    Integer boxed = 7;    // autoboxing
    int sum = boxed + 3; // unboxing then addition
    System.out.println(sum);           // 10
    System.out.println(safeUnbox(null)); // 0
  }
}
```

# Autoboxing and Unboxing

- What do these mean?
    - Autoboxing: Java automatically converts a primitive to its wrapper when an object is expected
    - Unboxing: Java automatically converts a wrapper to its primitive when a primitive is expected
- Where you will see this
    - Assigning int into an Integer variable or passing it to a method that expects Integer
    - Using an Integer where an int is needed triggers unboxing
- Rules to keep code safe
    - Never unbox a null wrapper; check for null and supply a default value
    - Use equals for comparing wrapper values; use == for primitives
    - Avoid unnecessary boxing in tight loops or simple arithmetic

```
public class BoxingDemo {
  static int safeUnbox(Integer maybe) {
    return (maybe != null) ? maybe : 0; // default
when null
  }
  public static void main(String[] args) {
    Integer boxed = 7;    // autoboxing
    int sum = boxed + 3; // unboxing then addition
    System.out.println(sum);           // 10
    System.out.println(safeUnbox(null)); // 0
  }
}
```

**That’s for today!**
**Any questions?**