

1. How to Print Output in Java



Key Concepts:

- `System.out.print()` → Prints text without moving to the next line.
- `System.out.println()` → Prints text and moves to the next line.
- `System.out.printf()` → Prints formatted output using format specifiers.



Syntax:

1. `System.out.print("text");`
2. `System.out.println("text");`
3. `System.out.printf("format", values);`
 - `%d` → integer
 - `%f` → float/double
 - `%s` → string
 - `%.2f` → float with 2 decimal places



Points to Remember:

- Every Java print statement ends with a semicolon `;`.
- `println()` adds a line break automatically.

- `printf()` gives control over output formatting.
- Use escape characters like `\n` (newline), `\t` (tab).



Example:

Q: Print "Hello" and "World" on separate lines

A:

```
System.out.println("Hello");  
System.out.println("World");
```

2. What is a Variable in Java?



Key Concepts:

- Variables are containers used to store data values.
- Java requires you to declare the type of variable before using it.
- The type defines what kind of data the variable can hold.



Formulae (Syntax):

1. `datatype variableName = value;`

2. Examples:

- `int age = 20;`
- `String name = "Rahul";`
- `float price = 99.5f;`

✓ Points to Remember:

- Variable names are case-sensitive and follow camelCase convention.
- Cannot start with numbers or use Java keywords.
- Use `int`, `double`, `char`, `boolean`, `String`, etc., based on the data type.
- Variables can be declared without assigning values initially.

Example:

Q: Declare a string and an integer variable, then print them

A:

```
String city = "Mumbai";  
  
int population = 20000000;  
  
System.out.println(city + " has population of " +
```

3. What are Conditionals in Java?

Key Concepts:

- Conditionals allow a program to make decisions based on certain conditions.
- Common conditional statements in Java are:

- `if`
- `if-else`
- `if-else if-else`
- `switch`

Syntax:

```
// if statement
if (condition) {
    // code block
}

// if-else
if (condition) {
    // block if true
} else {
    // block if false
}
```

```
}

// if-else if-else
if (condition1) {
    // block 1
} else if (condition2) {
    // block 2
} else {
    // default block
}

// switch statement
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // default code
}
```

4. What are Loops in Java?



Key Concepts:

- Loops are used to execute a block of code repeatedly.
- Java has three main types of loops:
 - `for` loop — when number of iterations is known.
 - `while` loop — when the condition is checked before the block runs.
 - `do-while` loop — executes the block at least once, then checks the condition.



Syntax:

```
// for loop
for (initialization; condition; update) {
    // code to run
}

// while loop
while (condition) {
    // code to run
}

// do-while loop
do {
    // code to run
} while (condition);
```

5. How to Take Input in Java?



Key Concepts:

- **Input:** Receiving data from the user during program execution.
- **Scanner Class:** A built-in Java class (`java.util.Scanner`) used to read input from the keyboard.
- **Object Creation:** Create a `Scanner` object to use its methods like `nextInt()`, `nextLine()`.



Syntax:

1. **Import Scanner:** `import java.util.Scanner;`
(at the top of the file)
2. **Create Scanner Object:** `Scanner sc = new Scanner(System.in);`
3. **Read Input:**
 - `int num = sc.nextInt();` → Reads an integer
 - `double val = sc.nextDouble();` → Reads a decimal
 - `String text = sc.nextLine();` → Reads a line of text



Points to Remember:

- Always import `Scanner` before using it.
- `System.in` connects the Scanner to the keyboard.
- After `nextInt()`, add `sc.nextLine()` to clear the buffer before reading a string.
- Close the Scanner with `sc.close();` when done (good practice).



Example:

Q: How do you take a user's name (string) and age (integer) as input and print them?

A:

```
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
        String name = sc.nextLine(); // Reads string

        System.out.print("Enter your age: ");
        int age = sc.nextInt(); // Reads integer

        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```



```
// Scenario: Integer first, then String
System.out.print("Enter your age: ");
age = sc.nextInt(); // Reads integer
sc.nextLine(); // Clears leftover newline

System.out.print("Enter your name: ");
name = sc.nextLine(); // Reads string





System.out.println("Name: " + name + ", Age: " + age);
sc.close();
    }
}
```



Functions in Java (aka Methods)

In Java, a **function** is called a **method**. It is a **block of code** that performs a specific task and runs only when called.

✓ Why Use Functions?

-  Reusability of code
 -  Modular design
 -  Cleaner and more readable code
 -  Easy to debug and test
-



Basic Syntax

```
returnType functionName(parameters) {  
    // code to execute  
    return value; // if returnType is not void  
}
```







Example

```
public int add(int a, int b) {  
    return a + b;  
}
```

```
// Calling the function  
int result = add(5, 3); // result = 8
```

Types of Functions in Java

Type	Description
 Parameterized	Accepts arguments
 Non-Parameterized	Doesn't take any parameters
 With Return	Returns a value
 Void (No Return)	Performs a task but returns nothing (<code>void</code>)



Example: All Variants

```
// 1. No parameters, no return
```

```
public void greet() {  
    System.out.println("Hello!");  
}
```

```
// 2. With parameters, no return
```

```
public void greetUser(String name) {  
    System.out.println("Hello, " + name + "!");  
}
```

```
// 3. No parameters, with return
```

```
public int getDefaultAge() {  
    return 18;  
}
```

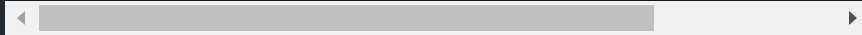
```
// 4. With parameters and return
```

```
public int square(int x) {  
    return x * x;  
}
```



Calling Methods

```
greet(); // Calls method witho  
greetUser("Alice"); // Passes argument  
int age = getDefaultAge(); // Captures returned  
System.out.println(square(4)); // Output: 16
```





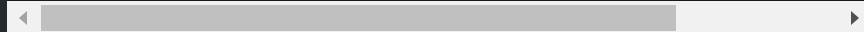
Access Modifiers

Modifier	Description
<code>public</code>	Accessible from anywhere
<code>private</code>	Accessible only within the same class
<code>protected</code>	Accessible in same package or subclass
(default)	Package-private (no keyword)

Static vs Non-Static Methods

- **Static Method:** Belongs to the class
- **Non-Static Method:** Belongs to an object instance

```
public static void show() { ... } // No need to cr  
public void display() { ... }    // Need object t
```









Return Statement

```
return value;
```

- Ends the function
 - Sends back result (if not `void`)
-



Best Practices

-  Use meaningful function names
 -  Keep functions small and focused
 -  Reuse logic through functions
 -  Document with comments and JavaDoc
-



Interview Tip:

"In Java, methods (functions) allow **modular programming**, making code more reusable, testable, and maintainable."

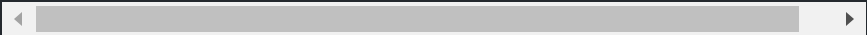


Number Systems in Programming (Java Focus)

In programming and computer science, a **number system** defines how numbers are represented and manipulated. Java and most other programming languages support **multiple number systems**, mainly:

Number System	Base	Digits Used	Common Use
Binary	2	0, 1	Low-level programming bitwise
Octal	8	0–7	Legacy systems
Decimal	10	0–9	Human-readable numbers
Hexadecimal	16	0–9 and	Memory addressing,

Number System	Base	Digits Used	Common Use
		A–F	colors

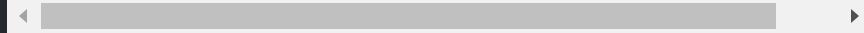


✓ Why Should Programmers Learn Number Systems?

- 🔧 Bitwise operations
 - 📦 Data compression and encoding
 - 🎨 Color representation (Hex)
 - 📍 Memory and address manipulation
 - 🧠 Understanding how the computer processes data
-

Java Support for Number Systems

```
int decimal = 100;      // Decimal
int binary  = 0b1101;   // Binary (prefix 0b)
int octal   = 0123;     // Octal (prefix 0)
int hex     = 0x1A3F;   // Hexadecimal (prefix 0x)
```





General Rules for Converting Number Systems

From	To	Rule / Steps
Decimal → Binary / Octal / Hex	Divide the number by 2 / 8 / 16 repeatedly. Write down the remainders in reverse order.	
Binary → Decimal	Multiply each bit by powers of 2 from right to left, then add.	
Octal → Decimal	Multiply each digit by powers of 8 from right to left, then add.	
Hex → Decimal	Multiply each digit by powers of 16 from right to left, then add (A=10 to F=15).	

From	To	Rule / Steps
Binary → Octal	Group bits in 3s (right to left), convert each group to octal digit.	
Binary → Hex	Group bits in 4s (right to left), convert each group to hex digit.	
Octal / Hex → Binary	Convert each digit into 3-bit (Octal) or 4-bit (Hex) binary.	



Conversion Example

◆ Decimal to Binary

Decimal: 13

$13 \div 2 = 6$, remainder = 1

$6 \div 2 = 3$, remainder = 0

$3 \div 2 = 1$, remainder = 1

$1 \div 2 = 0$, remainder = 1

Binary = 1101

◆ Binary to Decimal

Binary: 1101

$= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 8 + 4 + 0 + 1 = 13$



Java Methods for Conversion

```
Integer.toBinaryString(13); // "1101"
```

```
Integer.toOctalString(13); // "15"
```

```
Integer.toHexString(13); // "d"
```

```
Integer.parseInt("1101", 2); // 13
```

```
Integer.parseInt("15", 8); // 13
```

```
Integer.parseInt("d", 16); // 13
```




Summary Table

System	Base	Prefix	Java Example
Decimal	10	None	<pre>int x = 100;</pre>
Binary	2	<code>0b</code>	<pre>int x = 0b1010;</pre>
Octal	8	<code>0</code>	<pre>int x = 012;</pre>
Hexadecimal	16	<code>0x</code>	<pre>int x = 0x1F;</pre>

Real-Life Applications

Application	Number System
File permissions	Octal
IP/MAC addresses	Hexadecimal
Color Codes (#fff)	Hexadecimal
Bitmasking/flags	Binary
Calculations	Decimal

 Mastering number systems builds the foundation for understanding how data is stored, processed, and optimized in programming.



Arrays in Java – Complete Notes

In Java, an **array** is a **container object** that holds a fixed number of values of a **single data type**. Arrays are used to store multiple values in a **single variable**, instead of declaring separate variables for each value.

✓ Key Characteristics of Arrays

Feature	Description
Fixed Size	Size is set when the array is created and cannot change.
Zero-based Indexing	First element is at index <code>0</code> , last at <code>length - 1</code> .
Homogeneous Elements	All elements must be of the same data type.
Stored in Contiguous Memory	Array elements are stored next to each other in memory.

Array Declaration and Initialization

◆ Syntax

```
dataType[] arrayName;           // Declaration  
arrayName = new dataType[size]; // Memory allocati
```

◆ Combined Declaration and Allocation

```
int[] numbers = new int[5]; // Array of size 5
```


◆ Initialize with Values

```
int[] marks = {90, 85, 88, 76, 95};
```

Accessing and Modifying Elements

```
System.out.println(marks[0]); // Access first element
```

```
marks[2] = 100; // Modify 3rd element
```

 Accessing an index out of bounds will
throw `ArrayIndexOutOfBoundsException`.

Iterating Over Arrays

◆ Using **for** loop

```
for (int i = 0; i < marks.length; i++) {  
    System.out.println(marks[i]);  
}
```

◆ Using **for-each** loop

```
for (int mark : marks) {  
    System.out.println(mark);  
}
```



Array Properties

Property	Description
<code>length</code>	Returns size of array (no <code>()</code> like methods)
<code>index</code>	Starts from <code>0</code> and ends at <code>length - 1</code>

```
System.out.println(marks.length); // 5
```



Types of Arrays

1

One-Dimensional Array

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

2

Multi-Dimensional Array (Matrix)

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

matrix[0][0] = 1;

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```



Use Cases

- Storing student grades
 - Representing matrices
 - Data tables in games
 - Lookup tables
-

! Limitations of Arrays

Limitation	Alternative
Fixed size (non-resizable)	Use <code>ArrayList</code>
Can hold only one data type	Use <code>Object[]</code> or Collections
No built-in functions (e.g. sort, search)	Use utility classes like <code>Arrays</code>



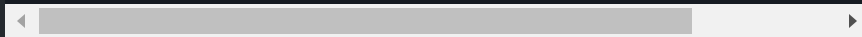
Utility Methods –

java.util.Arrays

```
import java.util.Arrays;

int[] arr = {5, 3, 9, 1};
Arrays.sort(arr); // Sorts the
System.out.println(Arrays.toString(arr)); // Print

int index = Arrays.binarySearch(arr, 3); // Binary
```





Common Interview Questions

Question	Concept Tested
Reverse an array	Looping logic
Find largest/smallest element	Conditional checking
Check for duplicates	Nested loops / HashSet
Sort an array	Sorting algorithms / Arrays.sort
Rotate array elements	Index manipulation



Mini Exercise

```
// Print sum of array elements  
  
int[] nums = {2, 4, 6, 8};  
  
int sum = 0;  
  
for (int n : nums) {  
    sum += n;  
}  
  
System.out.println("Sum = " + sum);
```



Arrays are the building blocks of data structures. Mastering them will give you a strong foundation for learning Lists, Stacks, Queues, and more!

How Arrays Are Stored in Memory in Java

In Java, arrays are **objects** stored in the **heap memory**, and they are accessed through **reference variables** stored in the **stack**. Let's understand this in detail.

◆ Components of Array Storage

When you declare and initialize an array:

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

Java stores the array in two parts:

Part	Memory Location	Description
Reference variable (<code>arr</code>)	Stack	Holds the reference (address) to the array
Actual array object	Heap	Contains array metadata and elements

◆ Memory Representation

```
int[] arr = {10, 20, 30, 40, 50};
```

Heap Memory (Contiguous Allocation for Elements):

Index	Address	Value
0	0x100	10
1	0x104	20
2	0x108	30
3	0x10C	40
4	0x110	50

- If `int` takes 4 bytes, each value is stored 4 bytes apart.
 - The reference variable `arr` (in the stack) points to the base address `0x100` of the array in the heap.
-

◆ Array Memory Layout Summary

[Stack]

+-----+

| arr: 0x100 |

+-----+

[Heap]

+-----+-----+-----+-----+

| 10 | 20 | 30 | 40 | 50 |

+-----+-----+-----+-----+

0x100 0x104 0x108 0x10C 0x110

Key Points



- Arrays are **objects** in Java, even if they store primitive types.
- The **length** property is stored with the array metadata in the heap.
- Java automatically **bounds-checks** arrays; accessing out-of-bounds throws `ArrayIndexOutOfBoundsException`.
- Arrays in Java are always **contiguously stored**, ensuring efficient access via index.



💡 Tip: Use `System.identityHashCode(arr)` to get the memory reference hash (not exact memory address) of the array.

◆ Example: Shared Reference Behavior in Arrays

```
int[] arr = new int[5];  
arr[0] = 33;  
arr[1] = 47;  
arr[2] = 59;  
arr[3] = 67;  
arr[4] = 98;  
  
System.out.print(arr[2]); // Output: 59  
  
int[] two = arr;           // 'two' now references  
two[2] = 200;  
  
System.out.print(arr[2]); // Output: 200
```

Explanation

-  `arr` and `two` both refer to the **same memory location** in heap.
-  When we assign `two = arr`, we are copying the **reference**, not the array itself.

-  Modifying `two[2] = 200` changes the value at index 2 in the original array too.
 -  That's why `arr[2]` also becomes `200`.
-



Memory Visualization

[Stack]

```
+-----+      +-----+
| arr      |-----> |      |
| two      |-----> | [Heap] |
+-----+      |-----|
                  | [0] = 33  |
                  | [1] = 47  |
                  | [2] = 200 |
                  | [3] = 67  |
                  | [4] = 98  |
                  +-----+
```

Key Takeaway

In Java, assigning one array to another does **not copy values**, it **copies the reference**, so both variables point to the **same memory block** in heap.

This concept is crucial when working with arrays and object references in Java!



Shared Reference Behavior of Arrays When Passed to a Function in Java

In Java, when you pass an array to a method, you're **passing the reference to the array object**, not a separate copy of the array. This means **modifications inside the method affect the original array**.

Example: Passing Array to a Method

```
public class Main {  
    public static void modifyArray(int[] arr) {  
        arr[1] = 999; // Modify index 1  
    }  
  
    public static void main(String[] args) {  
        int[] numbers = {10, 20, 30};  
  
        System.out.println("Before: " + numbers[1]  
        modifyArray(numbers);  
        System.out.println("After: " + numbers[1])  
    }  
}
```



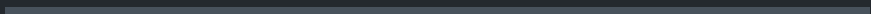
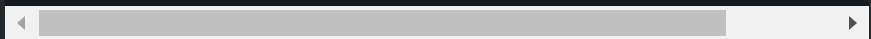
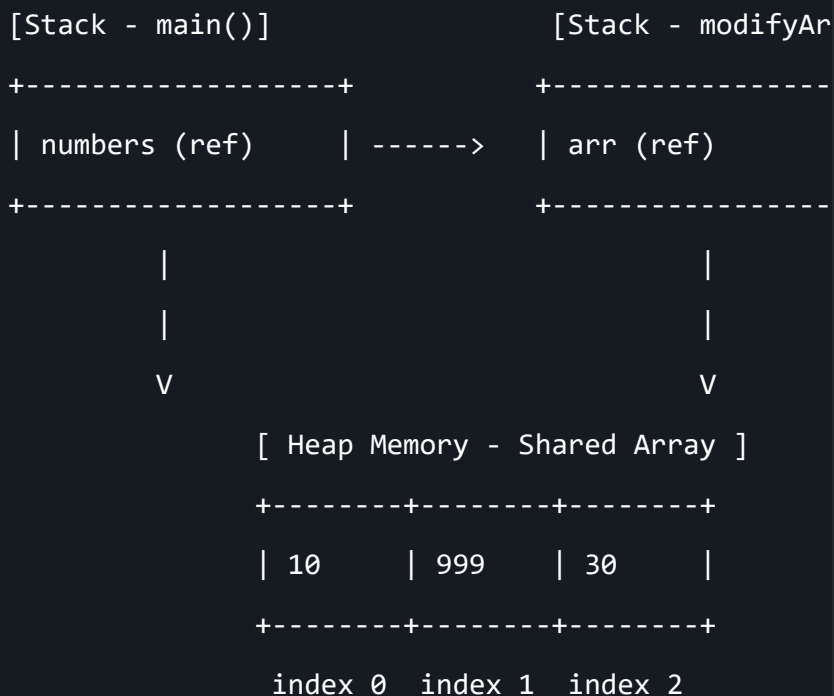


Explanation

Step	Action
1	<code>numbers</code> is declared and initialized in <code>main</code> .
2	<code>modifyArray(numbers)</code> passes the reference to the <code>modifyArray</code> method.
3	Inside the method, <code>arr[1] = 999</code> modifies the actual array in heap memory .
4	After the method call, <code>numbers[1]</code> is now <code>999</code> in the original array.



Memory Representation




✓ Key Points


- Arrays in Java are **passed by value**, but that value is a **reference** to the object.
 - Changes made inside the function reflect **outside the function**, as both point to the **same array**.
 - This is known as **shared reference behavior**.
-

Gotcha

If you reassign the reference inside the method (e.g., `arr = new int[]{1,2,3};`), it **won't affect** the original array because you're changing what the local reference points to — not the original object.

```
public static void modifyArray(int[] arr) {  
    arr = new int[]{1, 2, 3}; // This does NOT aff  
}
```

 To truly copy an array and avoid affecting the original, use `Arrays.copyOf()` or `array.clone()`.

 Use this knowledge to carefully manage side effects when passing arrays to functions.

Object References, Shallow Copy vs Deep Copy in Java

In Java, **objects** are not passed or assigned **directly**, but via **references**. This leads to behaviors like **shared modification**, especially with **mutable objects** like arrays or custom classes.

What is an Object Reference?

An **object reference** is a variable that **stores the memory address** of an object in the heap, not the object itself.

```
class Student {  
    String name;  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.name = "Alice";  
  
        Student s2 = s1; // s2 points to the same  
        s2.name = "Bob";  
  
        System.out.println(s1.name); // Output: Bo  
    }  
}
```



Explanation: `s1` and `s2` both point to the same memory location, so a change via `s2`

reflects in `s1`.



Shallow Copy

A **shallow copy** copies the reference of an object — **not its actual content**. So the original and copy share the **same inner objects**.



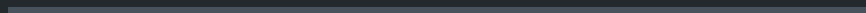
Example

```
class Student {  
    String name;  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        s1.name = "Alice";  
  
        Student s2 = s1; // Shallow copy  
  
        s2.name = "Bob";  
  
        System.out.println(s1.name); // Output: Bo  
    }  
}
```



Characteristics

Feature	Shallow Copy
Memory allocation	Shared
Performance	Faster
Side Effects	High
Suitable for	Immutable or simple objects





Deep Copy

A **deep copy** creates a **completely new copy** of the object and all its nested objects — no shared memory.



Example Using Constructor

```
class Student {  
    String name;  
  
    Student(String name) {  
        this.name = name;  
    }  
  
    // Deep copy constructor  
    Student(Student s) {  
        this.name = new String(s.name);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student("Alice");  
        Student s2 = new Student(s1); // Deep copy  
  
        s2.name = "Bob";  
    }  
}
```



```
        System.out.println(s1.name); // Output: A1  
    }  
}
```

Characteristics

Feature	Deep Copy
Memory allocation	Independent
Performance	Slower
Side Effects	None
Suitable for	Mutable or complex objects



Array Deep vs Shallow Example

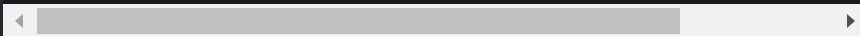
```
int[] original = {1, 2, 3};

// Shallow copy
int[] shallow = original;

// Deep copy
int[] deep = original.clone();

shallow[0] = 99;
deep[1] = 88;

System.out.println(Arrays.toString(original)); //
System.out.println(Arrays.toString(shallow)); //
System.out.println(Arrays.toString(deep));      //
```





When to Use What?

Use Case	Type
Simple, performance-critical task	Shallow Copy
Handling mutable or nested objects	Deep Copy
Preventing unintended changes	Deep Copy
Working with immutable objects	Shallow Copy



Key Takeaway: Java uses **reference semantics**. Understand when you're copying **data vs reference**, and use **deep copy** when isolation of data is essential.



Stacks in Java

A **stack** is a **linear data structure** that follows the **Last In, First Out (LIFO)** principle. This means that the **last element added** to the stack is the **first one to be removed**.

◆ Why Use a Stack?

- ✓ Supports **undo/redo** operations (e.g., text editors)
 - ✓ Manages function calls in **recursion**
 - ✓ Used in **expression evaluation** (e.g., parsing expressions)
 - ✓ Backtracking (e.g., **maze solving, browser history**)
-



Stack Operations

Operation	Description
<code>push(x)</code>	Adds element <code>x</code> to the top of the stack
<code>pop()</code>	Removes and returns the top element
<code>peek()</code>	Returns the top element without removing it
<code>isEmpty()</code>	Returns <code>true</code> if stack is empty
<code>size()</code>	Returns the number of elements in the stack

Implementing Stack in Java

Java provides **two ways** to implement a stack:

1 Using `Stack` Class (Java Collection Framework)

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println(stack.peek()); // 30
        System.out.println(stack.pop());  // 30
        System.out.println(stack.isEmpty()); // fa
    }
}
```

✓ **Pros:** Built-in, optimized

✗ **Cons:** Synchronized (slower for multi-threading)

2 Implementing Stack Using an Array (Manual Approach)

```
class StackArray {  
    private int[] arr;  
    private int top;  
    private int capacity;  
  
    public StackArray(int size) {  
        arr = new int[size];  
        capacity = size;  
        top = -1;  
    }  
  
    public void push(int x) {  
        if (top == capacity - 1) {  
            System.out.println("Stack Overflow");  
            return;  
        }  
        arr[++top] = x;  
    }  
}
```



```
public int pop() {
    if (top == -1) {
        System.out.println("Stack Underflow");
        return -1;
    }
    return arr[top--];
}

public int peek() {
    return (top == -1) ? -1 : arr[top];
}

public boolean isEmpty() {
    return top == -1;
}
}

public class Main {
    public static void main(String[] args) {
        StackArray stack = new StackArray(5);

        stack.push(10);
        stack.push(20);

        System.out.println(stack.peek()); // 20
        System.out.println(stack.pop());  // 20
        System.out.println(stack.isEmpty()); // fa
```

```
}  
}
```

✓ **Pros:** Faster, thread-safe

✗ **Cons:** Fixed size, needs resizing

Stack Using Linked List (Dynamic)

```
class Node {  
    int data;  
    Node next;  
}  
  
class StackLinkedList {  
    private Node top;  
  
    public StackLinkedList() {  
        this.top = null;  
    }  
  
    public void push(int x) {  
        Node newNode = new Node();  
        newNode.data = x;  
        newNode.next = top;  
        top = newNode;  
    }  
  
    public int pop() {  
        if (top == null) {  
            System.out.println("Stack Underflow");  
            return -1;  
        }  
    }  
}
```

```

        }

        int value = top.data;
        top = top.next;
        return value;
    }

    public int peek() {
        return (top == null) ? -1 : top.data;
    }

    public boolean isEmpty() {
        return top == null;
    }
}

public class Main {
    public static void main(String[] args) {
        StackLinkedList stack = new StackLinkedList();

        stack.push(10);
        stack.push(20);

        System.out.println(stack.peek()); // 20
        System.out.println(stack.pop());  // 20
        System.out.println(stack.isEmpty()); // false
    }
}

```

✓ **Pros:** Dynamic size, no overflow

✗ **Cons:** More memory usage (extra pointers)



Stack Applications

Application	Use Case
Function Calls	Call Stack in recursion
Undo/Redo	Text editors
Parentheses Matching	Syntax validation
Postfix & Prefix Evaluation	Expression parsing
DFS (Depth First Search)	Graph traversal



Key Takeaways

- 1 **Stack follows LIFO** (Last In, First Out).
- 2 Java provides `Stack<T>` **class**, but manual implementations offer more flexibility.
- 3 **Array-based stacks** are faster but have a fixed size.
- 4 **Linked list stacks** are dynamic but use extra memory.
- 5 Stacks are useful in **recursion, expression evaluation, and backtracking**.



Tip: Always use `try { pop(); } catch (EmptyStackException e) {}` when working with Java's `Stack` class to handle errors safely.

Infix, Postfix, and Prefix Notations in Java

In mathematical expressions, operators and operands can be arranged in different ways, leading to three main notations: **Infix**, **Postfix**, and **Prefix**. Understanding these notations is crucial for expression evaluation, parsing, and

conversion, especially in Java, where stacks are often used for such operations.

1. Infix Notation

Definition:

- In **infix notation**, the operator is placed **between** operands.
- This is the standard way humans write mathematical expressions.
- Example:

```
(3 + 5) * 2
```

Evaluation in Java:

- Infix expressions follow **operator precedence** and **associativity** rules.
- Java evaluates infix expressions directly using arithmetic operators and parentheses.
- **Example in Java:**

```
int result = (3 + 5) * 2; // result = 16  
System.out.println(result);
```

- **Limitations:**
 - Requires parsing and precedence handling when evaluated from a string.

- Cannot be easily processed by computers without additional logic.
-

2. Postfix Notation (Reverse Polish Notation - RPN)

Definition:

- In **postfix notation**, the operator is placed **after** the operands.
- No need for parentheses since the order of operations is unambiguous.
- Example:

```
3 5 + 2 *
```

This is equivalent to `(3 + 5) * 2`.

Evaluation in Java (Using Stack):

- Postfix expressions can be evaluated using a **stack**:
 - i. Scan the expression from left to right.
 - ii. Push operands onto the stack.
 - iii. When an operator is encountered, pop two operands, apply the operator, and push the result back.

- **Example in Java:**

```
import java.util.*;

public class PostfixEvaluation {
    public static int evaluatePostfix(String
        Stack<Integer> stack = new Stack<>());

    for (char ch : exp.toCharArray()) {
        if (Character.isDigit(ch)) {
            stack.push(ch - '0'); // Conv
        } else {
            int v2 = stack.pop();
            int v1 = stack.pop();
            switch (ch) {
                case '+': stack.push(v1 +
                case '-': stack.push(v1 -
                case '*': stack.push(v1 *
                case '/': stack.push(v1 /
            }
        }
    }
    return stack.pop();
}

public static void main(String[] args) {
    String postfix = "35+2*"; // (3+5)*2
    System.out.println(evaluatePostfix(po
```

```
}  
}
```

- **Advantages:**

- No need for precedence handling.
 - Easy to evaluate using stacks.
-

3. Prefix Notation (Polish Notation)

Definition:

- In **prefix notation**, the operator is placed **before** the operands.
- Like postfix, no parentheses are required.
- Example:

```
* + 3 5 2
```

This is equivalent to `(3 + 5) * 2`.

Evaluation in Java (Using Stack):

- Prefix expressions can be evaluated similarly to postfix:
 - i. Scan the expression **right to left**.
 - ii. Push operands onto the stack.
 - iii. When an operator is encountered, pop two operands, apply the operator, and push the result back.
- **Example in Java:**

```

import java.util.*;

public class PrefixEvaluation {
    public static int evaluatePrefix(String exp) {
        Stack<Integer> stack = new Stack<>();

        for (int i = exp.length() - 1; i >= 0; i--) {
            char ch = exp.charAt(i);
            if (Character.isDigit(ch)) {
                stack.push(ch - '0');
            } else {
                int v1 = stack.pop();
                int v2 = stack.pop();
                switch (ch) {
                    case '+': stack.push(v1 + v2);
                    case '-': stack.push(v1 - v2);
                    case '*': stack.push(v1 * v2);
                    case '/': stack.push(v1 / v2);
                }
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        String prefix = "*+352"; // (3+5)*2
        System.out.println(evaluatePrefix(prefix));
    }
}

```

```
}  
}
```

- **Advantages:**
 - No need for precedence handling.
 - Useful in **compilers and expression evaluation.**
-

Conversion Between Notations

Conversion	Algorithm Used
Infix → Postfix	Shunting-yard algorithm (Uses stack)
Infix → Prefix	Reverse infix → Convert to postfix → Reverse result
Postfix → Infix	Process using stack, insert operators at correct places
Prefix → Infix	Process right-to-left using stack

- **Example: Converting Infix to Postfix in Java**

```
import java.util.*;

public class InfixToPostfix {
    public static int precedence(char ch) {
        if (ch == '+' || ch == '-') return 1;
        if (ch == '*' || ch == '/') return 2;
        return -1;
    }
}
```

```

public static String infixToPostfix(String exp) {
    Stack<Character> stack = new Stack<>();
    StringBuilder result = new StringBuilder();

    for (char ch : exp.toCharArray()) {
        if (Character.isDigit(ch)) {
            result.append(ch);
        } else if (ch == '(') {
            stack.push(ch);
        } else if (ch == ')') {
            while (!stack.isEmpty() && stack.peek() != '(')
                result.append(stack.pop());
            stack.pop(); // Remove '('
        } else {
            while (!stack.isEmpty() && precedence(stack.peek()) > precedence(ch))
                result.append(stack.pop());
            stack.push(ch);
        }
    }

    while (!stack.isEmpty())
        result.append(stack.pop());

    return result.toString();
}

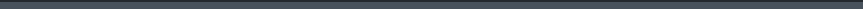
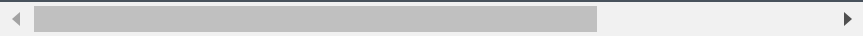
public static void main(String[] args) {

```

```
String infix = "3+5*2";  
System.out.println(infixToPostfix(inf  
    }  
}
```

Comparison Table

Notation	Expression Example	Evaluation Complexity	Ease of Use
Infix	(3 + 5) * 2	Medium (Handles precedence)	Easiest for humans
Postfix	3 5 + 2 *	Fast (Stack-based)	Harder to write manually
Prefix	* + 3 5 2	Fast (Stack-based)	Harder to write manually



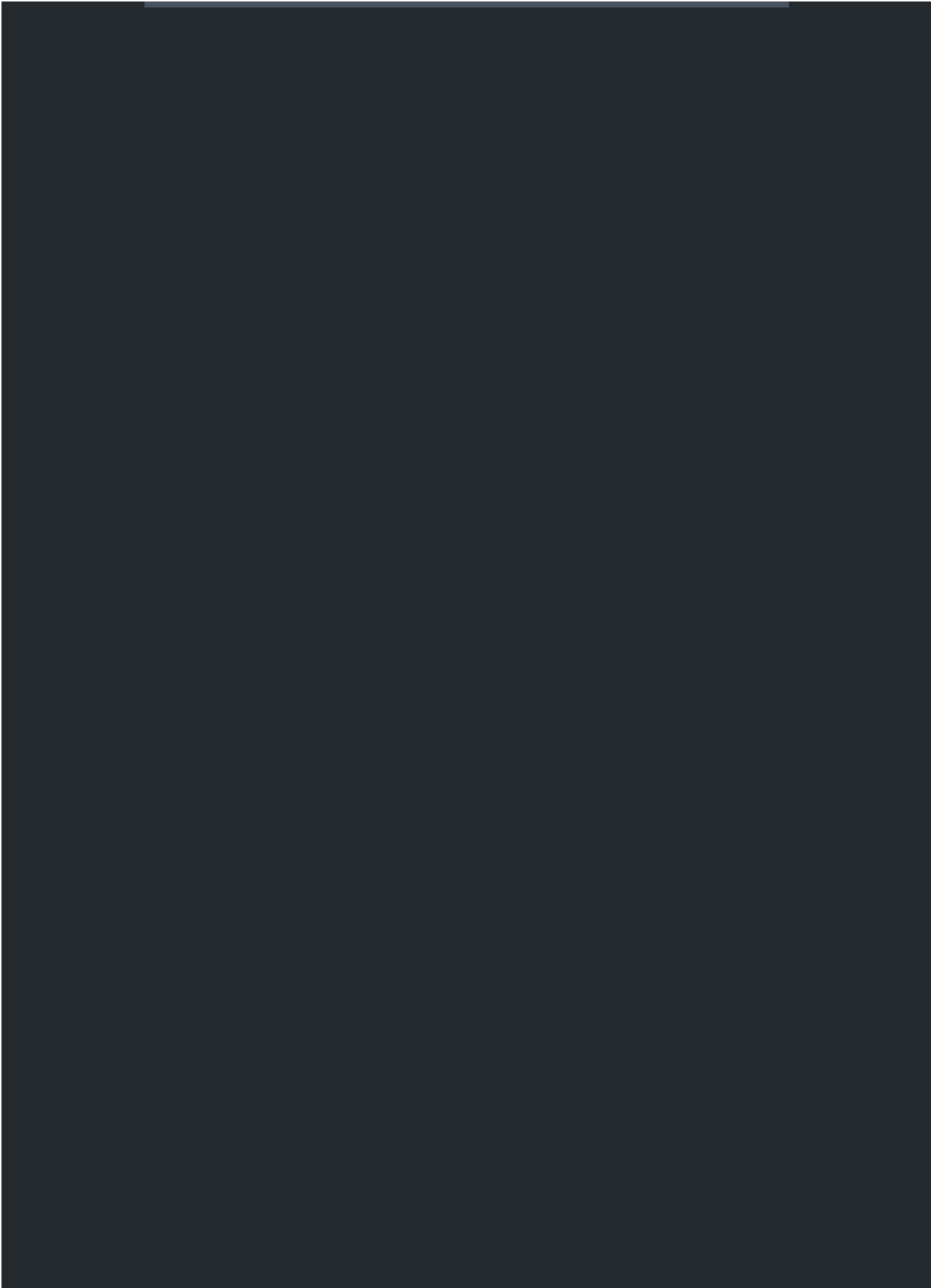
Conclusion

- **Infix notation** is human-friendly but requires precedence handling.
- **Postfix notation** is easier to evaluate using stacks, making it suitable for **compilers** and **calculators**.
- **Prefix notation** is useful in **recursive computations** and **expression trees**.

Java provides powerful **stack-based solutions** for evaluating and converting expressions between these notations, making it a core concept in **data structures, algorithms, and compilers**. 🚀

Fundamental Conversions Between Strings, Characters, and Numbers in Java

Understanding how to efficiently convert between **strings, characters, and numeric values** is crucial for handling data operations in Java. This note establishes a **common base** for these conversions, ensuring a structured understanding for all future operations.



1. Converting a Numeric Character in a String to an Integer

💡 Subtract `'0'` from a character digit to get its integer value.

Why?

- Characters are stored as **ASCII/Unicode values**.
- `'0'` (zero character) has an ASCII value of **48**.
- Any digit character `'0'` to `'9'` has a corresponding ASCII value from **48 to 57**.
- Subtracting `'0'` extracts the actual numeric value.

Example:

```
char digit = '7';  
int num = digit - '0'; // '7' (ASCII 55) - '0' (AS  
System.out.println(num); // Output: 7
```

Usage:

- ✓ Extracting integer values from numeric characters in strings.
 - ✓ Efficient for handling **single-digit** character conversions.
-

2. Converting a String Representation of a Number to an Integer

💡 Use `Integer.parseInt(str)` or `Integer.valueOf(str)`.

Example:

```
String numStr = "123";  
  
int number = Integer.parseInt(numStr); // Convert  
System.out.println(number); // Output: 123
```

- ✓ Works for **multi-digit** numbers.
 - ✓ `Integer.valueOf(str)` returns an `Integer` object instead of `int`.
-

3. Converting a Single Digit Integer to a Character

💡 Add `'0'` to an integer to get its character equivalent.

Why?

- Just as subtraction (`'7' - '0'`) extracts a numeric value,
- Adding `'0'` shifts an integer into its ASCII character range.

Example:

```
int num = 5;
char digitChar = (char) (num + '0'); // 5 + ASCII
System.out.println(digitChar); // Output: '5'
```

✓ Efficient for **single-digit numbers**.

4. Converting Any Number to a String

💡 Concatenate with `""` or use

`String.valueOf(num)`.

Examples:

```
int num = 123;

String str1 = num + ""; // Implicit conversion us
String str2 = String.valueOf(num); // Explicit con

System.out.println(str1); // Output: "123"
System.out.println(str2); // Output: "123"
```

✓ Works for **all numeric types** (`int`, `double`, `float`, etc.)

✓ **Preferred:** `String.valueOf(num)`, as it avoids unnecessary concatenation.

5. Converting a Character to a String

💡 Concatenate with `""` or use

`Character.toString(ch)` .

Example:

```
char ch = 'A';  
String str1 = ch + ""; // Implicit conversion  
String str2 = Character.toString(ch); // Explicit  
  
System.out.println(str1); // Output: "A"  
System.out.println(str2); // Output: "A"
```

✓ Useful for handling **single characters** in **string operations**.

6. Converting a String to a Character Array

💡 Use `toCharArray()` to break a string into individual characters.

Example:

```
String word = "Hello";  
char[] charArray = word.toCharArray();  
  
System.out.println(Arrays.toString(charArray)); //
```

✓ Useful for iterating over characters in a string.

Conclusion: Universal Conversion Rules

Conversion	Approach
String → Integer	<code>Integer.parseInt(str)</code>
String → Character Array	<code>str.toCharArray()</code>
Single Character → Integer	<code>ch - '0'</code>
Integer → Single Character	<code>(char) (num + '0')</code>
Number → String	<code>num + ""</code> OR <code>String.valueOf(num)</code>
Character → String	<code>ch + ""</code> OR <code>Character.toString(ch)</code>

Key Takeaways:

1. **Subtract** `'0'` to convert a numeric character to an integer.
2. **Add** `'0'` to convert an integer to a numeric character.

3. **Concatenation** (`"" +`) is a quick way to convert any number or character to a string.
4. **Use** `String.valueOf()` for efficient numeric-to-string conversion.
5. **Use** `toCharArray()` for character-level string processing.

This foundational understanding will help in **string manipulations, numerical operations, and type conversions** across Java programs.

