Java Basics

Contents

1. Introduction to Java

Java is a powerful, high-level, object-oriented programming language widely used for building applications that are secure, robust, and platform-independent

Here are a few reasons why Java is a good choice for beginners and professionals:

- **Platform Independence:** Java code can run on any machine with a Java Virtual Machine (JVM), making it highly portable.
- **Object-Oriented:** Java encourages modular programming and code reuse, making it easier to manage and scale.
- **Rich API and Libraries:** Java provides an extensive API and libraries that simplify complex tasks.
- **Security:** Java has built-in security features, making it a preferred choice for developing secure applications.
- **Community and Support:** Java has a vast community and extensive documentation.

Java Use cases: -

1. **Desktop Apps**:- Some GUI toolkits in java
   - AWT(Abstract Windowing Tool)
   - Swing
   - JavaFX(advanced, have animation too)

2. **Enterprise Java**: - Java EE(Enterprise Edition), advanced application with web and database etc. Java EE is set of enterprise applications geared towards enterprise application, so that you don't need hundreds of application. Key components are:-
   - **Java Server Pages (JSP):** Used to create dynamic web pages with embedded Java code, allowing for server-side processing and presentation logic.
   - **Servlets:** Java classes that handle HTTP requests and responses, enabling dynamic web content generation.
   - **Enterprise JavaBeans (EJBs):** Components that manage business logic, including transaction handling, security, and persistence, often used for complex business operations.
   - **JPA (Java Persistence API):** A standard for managing object persistence in relational databases, simplifying database interactions.
   - **JAX-RS (Java API for RESTful Web Services):** Enables the development of RESTful web services for interacting with applications via HTTP requests.
   - **JMS (Java Message Service):** Facilitates asynchronous messaging between applications using message queues.

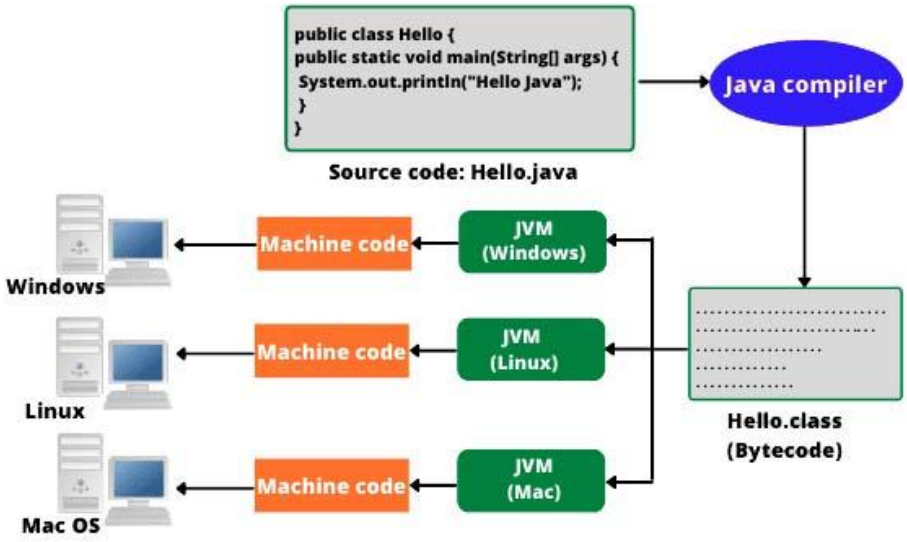
3. **Java Mobile Development**: - Using Android

4. **Java in cloud:** - web application development using microservices with frameworks like spring boot.

How Java code is executed?

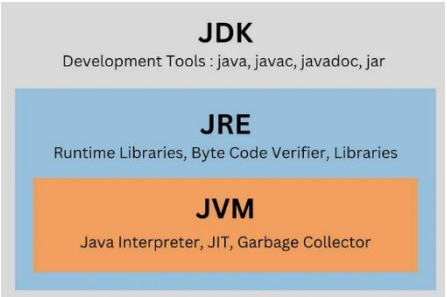
Java code execution involves two main steps: compilation and interpretation:

- **Compilation**: The Java compiler (javac) translates the human-readable Java source code (.java file) into platform-independent bytecode, stored in a .class file.
- **Execution**: The Java Virtual Machine (JVM) reads the bytecode and converts it into machine-specific instructions using the Interpreter and Just-In-Time (JIT) compiler. This allows the program to run on any operating system with a JVM, ensuring Java's "write once, run anywhere" principle.



JDK vs JRE vs JVM: -

1. JDK is for developers to create Java applications.
2. JRE is for users to run Java applications.
3. JVM is the engine that powers Java applications, making them platform-independent.



JDK (Java Development Kit): Used for developing Java applications.

Components:

- Compiler (javac): Translates Java source code into bytecode.
- Interpreter (java): Executes the bytecode.
- Debugger: Helps identify and fix errors in code.
- Libraries (JAR files): Pre-written code for common tasks.
- Documentation: Provides information on Java APIs and language features.

JRE (Java Runtime Environment): Used for running Java applications.

Components:

- JVM (Java Virtual Machine): The core component that executes bytecode.
- Class libraries: Provides essential classes for input/output, networking, and other functionalities.

JVM (Java Virtual Machine): Executes Java bytecode

Key Features:

- Platform Independence: Allows Java programs to run on any platform with a JVM.
- Memory Management: Automatically handles memory allocation and deallocation.
- Security: Provides security features to protect against malicious code.
- Garbage Collection: Automatically reclaims unused memory.

First Java code: -

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

2. Basic Java Concepts

Statement

A statement in Java represents a complete unit of execution. Examples include assignment, method calls, and control flow.

```
public class StatementsExample {
    public static void main(String[] args) {
        int x = 10;                 // Assignment statement
        System.out.println(x);      // Method call statement

        if (x > 5) {                // Conditional statement
            System.out.println("x is greater than 5");
        }
    }
}
```

Comments

Comments are notes in the code that are ignored by the compiler, used to explain code or disable it during debugging.

Types of Comments

- Single-line Comment: Starts with //.
- Multi-line Comment: Starts with /* and ends with */.
- Documentation Comment: Starts with /** and used for generating documentation.

```
public class CommentsExample {
    public static void main(String[] args) {
        // This is a single-line comment
        System.out.println("Single-line comment demo");

        /* This is a multi-line comment
           explaining the code below */
        System.out.println("Multi-line comment demo");

        /** This is a documentation comment
         * used for generating JavaDocs. */
        System.out.println("Documentation comment demo");
    }
}
```

Variables

Variables are containers that store data values. Every variable in Java has a type, which defines the kind of data it can hold.
Variable naming convention: camelCase

Data types

Java is a statically-typed language, which means variables must be declared with a specific type. Data types determine the kind of values that a variable can hold.

Types of Variables

- Primitive Variables: Store basic data types like int, float, char, etc.
- Reference Variables: Store addresses of objects.

Primitive Data type

Type	Size	Default Value	Example Values
byte	1 byte (8 bits)	0	-128 to 127
short	2 bytes	0	-32,768 to 32,767
int	4 bytes	0	-2,147,483,648 to 2,147,483,647
long	8 bytes	0L	-2^{63} to $2^{63}-1$
float	4 bytes	0.0f	3.4e-038 to 3.4e+038
double	8 bytes	0.0	1.7e-308 to 1.7e+308
char	2 bytes (UTF-16)	'\u0000'	'a', '1', '$', '\u0000'
boolean	1 bit	false	true, false

```
public class DataTypesExample {
    public static void main(String[] args) {
        byte b = 10;
        int i = 100;
        long l = 100000L;
        float f = 10.5f;
        double d = 99.99;
        char c = 'A';
        boolean flag = true;

        System.out.println("Byte: " + b);
        System.out.println("Int: " + i);
        System.out.println("Long: " + l);
        System.out.println("Float: " + f);
        System.out.println("Double: " + d);
        System.out.println("Char: " + c);
        System.out.println("Boolean: " + flag);
    }
}
```

Reference Data type
Reference types store references to objects or arrays. They include:

- Class Types: String, Integer.
- Interface Types: Runnable, List.
- Array Types: int[], String[].

```
public class ReferenceExample {
    public static void main(String[] args) {
        String name = "John";
        int[] numbers = {1, 2, 3};

        System.out.println("Name: " + name);
        System.out.println("First Number: " + numbers[0]);
    }
}
```

Operators

Operators perform operations on variables and values.

Types of Operators

- **Arithmetic Operators**: +, -, *, /, %.
- **Relational Operators**: ==, !=, >, <, >=, <=.
- **Logical Operators**: &&, ||, !.
- **Assignment Operators**: =, +=, -=, *=, /=.
- **Unary Operators**: ++, --, +, -.

```
public class OperatorsExample {
    public static void main(String[] args) {
        int a = 10, b = 20;

        // Arithmetic Operators
        System.out.println("Addition: " + (a + b));  //30
        System.out.println("Subtraction: " + (b - a));  //10

        // Relational Operators
        System.out.println("Is a equal to b? " + (a == b)); //false

        // Logical Operators
        System.out.println("Logical AND: " + (a > 5 && b > 15)); //true

        // Unary Operators
        System.out.println("Increment a: " + (++a)); //11
        System.out.println("Decrement b: " + (--b)); //19
    }
}
```

Operator Precedence

Operator precedence determines the order in which operators are evaluated in expressions.

The table below lists operators in order of precedence (higher precedence first):

Precedence Level	Operators	Associativity
1 (Highest)	`()`, `[]`, `.`, `::`	Left-to-right
2	`++`, `--` (postfix)	Left-to-right
3	`++`, `--` (prefix), `+`, `-` (unary), `~`, `!`	Right-to-left
4	`*`, `/`, `%`	Left-to-right
5	`+`, `-`	Left-to-right
6	`<<`, `>>`, `>>>`	Left-to-right
7	`<`, `<=`, `>`, `>=`, `instanceof`	Left-to-right
8	`==`, `!=`	Left-to-right
9	`&`	Left-to-right
10	`^`	Left-to-right
11	`	`	`	`
12	`&&`	Left-to-right
13	`	`	
14	`?:` (ternary)	Right-to-left
15	`=`, `+=`, `-=`, `*=`, `/=`, etc.	Right-to-left

Associativity

Associativity determines the direction in which operators of the same precedence are evaluated:

Left-to-right: Most operators (e.g., arithmetic, logical).
Right-to-left: Unary, assignment, and ternary operators

```java
//left to right Associativity
public class AssociativityExample {
    public static void main(String[] args) {
        int result = 10 - 5 + 2; // Evaluated as (10 - 5) + 2
        System.out.println("Result: " + result); // Output: 7
    }
}

//right to left Associativity
public class AssociativityExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int c = a = b; // Evaluated as a = (b)
        System.out.println("a: " + a); // Output: 20
        System.out.println("c: " + c); // Output: 20
    }
}
```

Control flow (if-else)

Syntax:-

```
if (condition) {
    // Code if condition is true
} else {
    // Code if condition is false
}
```

```
public class IfElseExample {
    public static void main(String[] args) {
        int number = 10;

        if (number % 2 == 0) {
            System.out.println(number + " is even");
        } else {
            System.out.println(number + " is odd");
        }
    }
}
```

Nested if-else

```
if (condition1) {
    // Executes if condition1 is true
    if (condition2) {
        // Executes if condition2 is also true
    } else {
        // Executes if condition2 is false
    }
} else {
    // Executes if condition1 is false
}
```

```
public class NestedIfElseExample {
    public static void main(String[] args) {
        int number = 25;

        if (number > 0) { // Outer if
            System.out.println("The number is positive.");

            if (number % 2 == 0) { // Inner if
                System.out.println("The number is even.");
            } else {
                System.out.println("The number is odd.");
            }

        } else {
            System.out.println("The number is negative.");
        }
    }
}
```

Loops

Loops execute a block of code repeatedly based on a condition.

1. For loop

```
for (initialization; condition; increment/decrement) {
    // Code block
}
```

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
        }
    }
}
```

2. while loop

```
while (condition) {
    // Code block
}
```

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("Count: " + i);
            i++;
        }
    }
}
```

3. do-while loop

```
do {
    // Code block
} while (condition);
```

```
public class DoWhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Count: " + i);
            i++;
        } while (i <= 5);
    }
}
```

3. Type Conversion

Type conversion refers to changing the type of a variable from one data type to another. In Java, type conversion occurs in two ways:

- **Implicit Type Conversion** (Type Promotion): Automatically performed by the Java compiler.
- **Explicit Type Conversion** (Type Casting): Manually specified by the programmer.

Implicit type conversion (Widening)

This type of conversion happens automatically when:

- Data is converted from a smaller data type to a larger data type.
- No data loss occurs during the conversion.

Smaller Type	Larger Type
byte	short
short	int
int	long
long	float
float	double

```
public class ImplicitConversion {
    public static void main(String[] args) {
        int num = 100;
        double result = num; // int is automatically converted to double

        System.out.println(num);  //100
        System.out.println(result); //100.0
    }
}
```

When performing operations with mixed data types, Java automatically promotes smaller types to the largest type in the operation.

```
public class NumericPromotion {
    public static void main(String[] args) {
        int num = 10;
        double result = num + 5.5; // int promoted to double

        System.out.println("Result: " + result); //Result: 15.5
    }
}
```

A char is treated as an integer (ASCII value) when used in numeric operations.

```java
public class CharConversion {
    public static void main(String[] args) {
        char ch = 'A';
        int asciiValue = ch; // char to int

        System.out.println("Character: " + ch);    //Character: A
        System.out.println("ASCII Value: " + asciiValue);  //ASCII Value: 65
    }
}
```

Explicit type conversion (Narrowing)

This type of conversion is done manually by the programmer and involves converting data from a larger data type to a smaller data type. Since there is a possibility of data loss, explicit casting is required using parentheses ().

Larger Type	Smaller Type
double	float
float	long
long	int
int	short
short	byte

Syntax: - `type smallerTypeVariable = (smallerType) largerTypeVariable;`

```java
public class ExplicitConversion {
    public static void main(String[] args) {
        double value = 100.99;
        int result = (int) value; // Explicitly casting double to int

        System.out.println(value);  //100.99
        System.out.println(result); //100
    }
}
```

Note: Boolean type cannot be converted to any other type or vice versa.

Conversion between String and Primitive data types: -

We can use Wrapper class methods to convert String to primitive data types

Conversion	Method
String → int	`Integer.parseInt(String)`
String → double	`Double.parseDouble(String)`
String → float	`Float.parseFloat(String)`
String → long	`Long.parseLong(String)`

```java
public class StringToPrimitive {
    public static void main(String[] args) {
        String str = "911";    //if it is "ab" > will get NumberFormatException
        int num = Integer.parseInt(str);

        System.out.println(str + " " + str.getClass().getName());
        System.out.println(num + " " + ((Object)num).getClass().getName());
    }
}
//911 java.lang.String
//911 java.lang.Integer
```

To Convert Primitive type to String, we can use:

- String.valueOf() method.
- Concatenation it with an empty string (+ "").

```java
public class PrimitiveToString {
    public static void main(String[] args) {
        int num = 123;
        String str = String.valueOf(num);

        System.out.println("Integer value: " + num);
        System.out.println("Converted String: " + str);
    }
}
//Integer value: 123
//Converted String: 123
```

4. Input and Output

Output

Java uses the System.out object to display output to the console.

Method	Description
print()	Outputs the text without a newline.
println()	Outputs the text with a newline at the end.
printf()	Formats and prints the output according to a format string.

```
public class PrintingExample {
    public static void main(String[] args) {
        // example of print(): output in same line, even without space
        System.out.print("Java");
        System.out.println("Coder");
        //Output: JavaCoder

        //example of println(): output by adding newline at end
        System.out.println("Hello Bro");
        System.out.println("How are you?");
        //output: Hello Bro
        //        How are you?
    }
}
```

printf() allows formatted output

Placeholder	Description
%d	Integer
%f	Floating-point number
%s	String
%c	Character

```
public class PrintingExample {
    public static void main(String[] args) {
        String name = "Roger";
        int age = 26;
        double salary = 5000.25;

        System.out.printf("%s of age %d earns salary of %.2f", name, age,
salary);
        //Roger of age 26 earns salary of 5000.25
    }
}
```

Escape characters: -

Escape Character	Description	Example	Output
`\n`	Newline	`System.out.println("Hello\nWorld!");`	Hello World!
`\t`	Tab	`System.out.println("A\tB");`	A B
`\\`	Backslash	`System.out.println("\\Java");`	\Java
`\"`	Double Quote	`System.out.println("\"Java\"");`	"Java"

Input

The Scanner class is the most commonly used way to read input from the console. It is part of the java.util package.

Methods of Scanner class

Method	Purpose
nextInt()	Reads an integer.
nextDouble()	Reads a double.
nextLine()	Reads an entire line as a string.
next()	Reads a single word.
nextBoolean()	Reads a boolean value.

```
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.print("Enter your salary: ");
        double salary = scanner.nextDouble();

        System.out.println("\n--- User Details ---");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

Output: -

```
Enter your name: John wick
Enter your age: 35
Enter your salary: 1000000

--- User Details ---
Name: John wick
Age: 35
Salary: 1000000.0
```

Note: - When using nextInt() or nextDouble(), a newline character is left in the buffer. To avoid issues when using nextLine() after these methods, consume the leftover newline with scanner.nextLine(). Normally happens while inputting String after int/double.

Issue:-

```java
public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.print("Enter your age: ");
        String name = scanner.nextLine();

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

Unable to input string

```
Enter your age: 35
Enter your Name: --- User Details ---
Name:
Age: 35
```

Fix:-

```java
public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        scanner.nextLine();   //fix
        System.out.print("Enter your Name: ");
        String name = scanner.nextLine();

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

5. Arrays

An array is a container object that holds a fixed number of values of a single data type. The length of an array is established when the array is created, and cannot be changed. (Arrays are immutable and not thread safe)

Declaration

```
// way to declare the array
int[] arr;

//ways to initialize the array
int[] arr = new int[4];

int[] arr = {1,2,3,4};
```

Default Values of Array Elements

- Numeric arrays are initialized to 0.
- boolean arrays are initialized to false.
- Reference type arrays (like String[]) are initialized to null.

Basic functions

```
public class Array {
   public static void main(String args[]){
      int[] arr = {1,2,3,4};

      //way to access the elements
      System.out.println("Second Element of array is "+ arr[1]);

      //Modifying the value
      arr[0] = 24;

      //printing the array
      for(int i=0; i< arr.length; i++){  //output 24 2 3 4
         System.out.println(arr[i]);
      }

      //For each loop
      for(int ele:arr){
         System.out.println(ele);  //output 24 2 3 4
      }
   }
}
```

Taking array as input from user

```
import java.util.*;

public class Array2 {
   public static void main(String args[]){
      Scanner scan = new Scanner(System.in);
      int size = scan.nextInt();

      int[] arr = new int[size];
      //Getting array input from users
      for(int i=0; i<size; i++){
         arr[i] = scan.nextInt();
      }
      //Printing array
      for(int ele: arr){
         System.out.println(ele);
      }

   }
}
```

2D Arrays

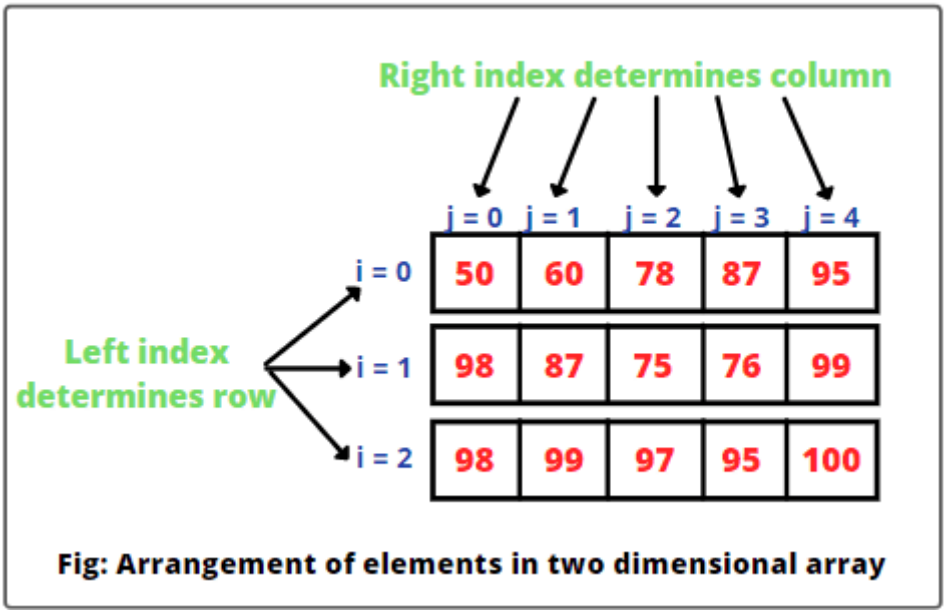


Fig: Arrangement of elements in two dimensional array

```
int[][] matrix = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
};
System.out.println(matrix[1][2]); // Output:  6
```

Array Utility class

Java provides the Arrays class with useful methods:

- sort(): Sorts the array in ascending order.
- binarySearch(): Searches for a specified value in a sorted array.
- copyOf(): Copies elements from an array.

```
import java.util.Arrays;

public class LearnArrayUtil {
   public static void main(String[] args) {

      int[] arr = {5, 2, 8, 1};
      Arrays.sort(arr); // Sorts array

      //Convert array to string to print in 1 line
      System.out.println(Arrays.toString(arr)); // Output: [1, 2, 5, 8]

      int[] arr2 = {4, 3, 7};
      int[] arr3 = {4, 3, 7};
      System.out.println(Arrays.compare(arr,arr2));   //-1 arr is larger than arr2
      System.out.println(Arrays.compare(arr2,arr3));  //0 > arrays are equal
   }
}
```

6. Strings

- A String is a sequence of characters.
- Strings in Java are objects of the String class, which is part of java.lang package.
- Strings are immutable, meaning their content cannot be changed after they are created.

```
public class LearnString {
   public static void main(String[] args) {

      //Create string using literal
      String s1 = "Hello";

      //Creating string using new Keyword
      String s2 = new String("World");

      System.out.println(s1 + " " + s2);

   }
}
```

How String is stored in memory

- Java uses a String Pool to optimize memory (The String Pool is a special area in the Heap memory where Java stores string literals to optimize memory usage).
- If two strings have the same content and are created using literals, they share the same memory location in the pool.
- Strings created using the new keyword are stored in the heap.

```
public class LearnString {
   public static void main(String[] args) {

      String s1 = "Java";
      String s2 = "Java";
      String s3 = new String("Java");
      String s4 = new String("Java");

      System.out.println(s1 == s2); // true (both refer to the same object in the String pool)
      System.out.println(s1 == s3); // false (different objects in memory)
      System.out.println(s3 == s4); // false (different objects in memory)

   }
}
```

String Methods

Method	Description	Example
length()	Returns the length of the string	"Hello".length() → 5
charAt(int index)	Returns the character at the specified index	"Java".charAt(1) → 'a'
substring(int start)	Extracts a substring from the string	"Hello".substring(2) → "llo"
substring(int start, int end)	Extracts a substring from start to end index	"Hello".substring(1, 4) → "ell"
equals(String s)	Compares strings for equality	"Java".equals("java") → false
equalsIgnoreCase(String s)	Compares strings, ignoring case	"Java".equalsIgnoreCase("java") → true
toLowerCase()	Converts the string to lowercase	"JAVA".toLowerCase() → "java"
toUpperCase()	Converts the string to uppercase	"java".toUpperCase() → "JAVA"
trim()	Removes leading/trailing whitespace	" Java ".trim() → "Java"
replace(oldChar, newChar)	Replaces characters in a string	"Hello".replace('l', 'p') → "Heppo"

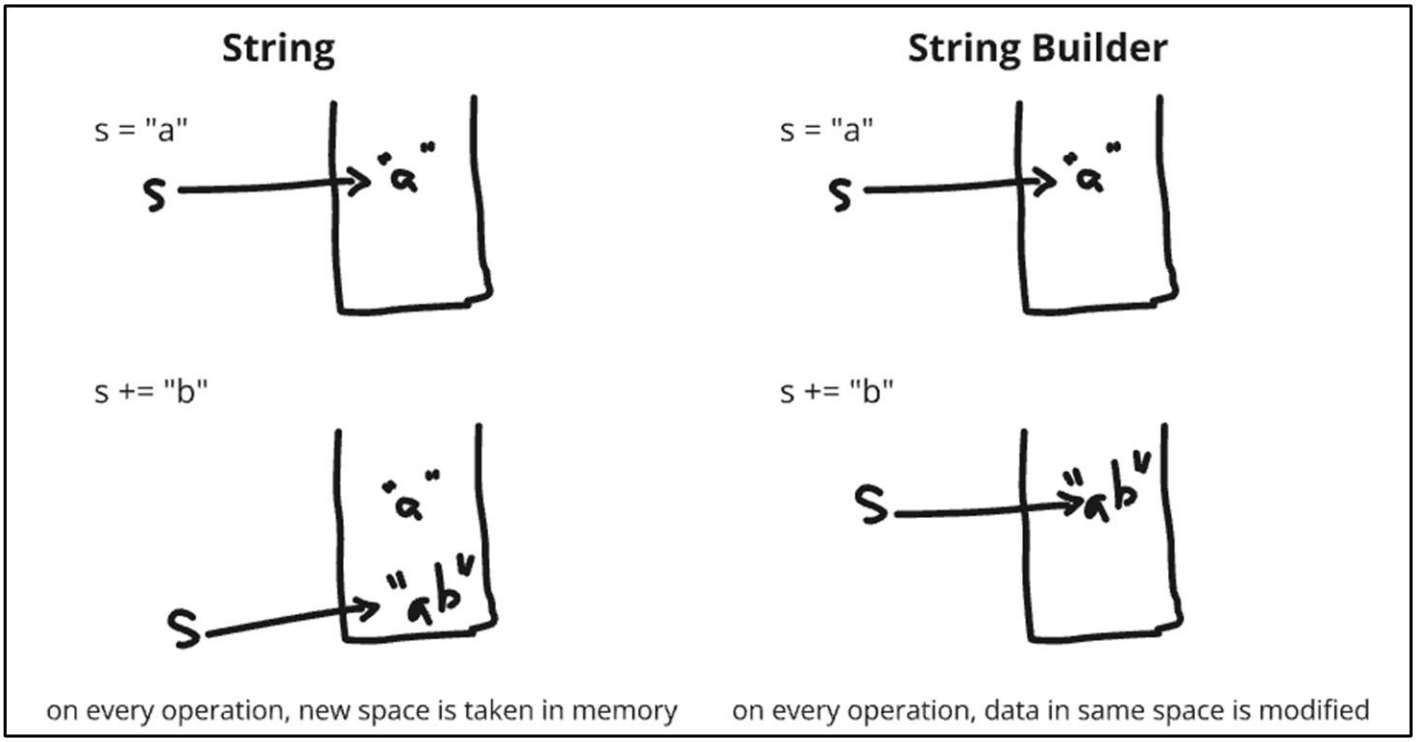
```
public class LearnString {
   public static void main(String[] args) {
      String str = "Java Programming";
      String str2= new String("Java Programming");

      System.out.println("Length: " + str.length());   //16
      System.out.println("Substring: " + str.substring(2, 8));  //va Pro
      System.out.println("Substring2: " + str.substring(5)); // Programming
      System.out.println("Replaced: " + str.replace('a', 'o')); //Jovo Programming
      System.out.println("Uppercase: " + str.toUpperCase());  //JAVA PROGRAMMING
      System.out.println(str.equals(str2)); // true(as .equals check for content)
   }
}
```

String Builder

The StringBuilder class in Java is a mutable sequence of characters used for efficiently creating and manipulating strings. Unlike the String class, which creates new objects for every modification, StringBuilder modifies the existing object.

How String and String Builder works: -



- Strings in Java are immutable. Every time you modify a string (e.g., concatenation or replacement), a new String object is created, which is both memory and time-intensive.
- StringBuilder, being mutable, performs operations directly on the same object, making it faster and more memory-efficient for repetitive string modifications.

Aspect	String	StringBuilder
Mutability	Immutable	Mutable
Performance	Slower for multiple modifications	Faster for multiple modifications
Memory Usage	Creates new objects for changes	Modifies the same object
Thread-Safety	Thread-safe	Not thread-safe

Key Methods of String Builder

Method	Description
append(String s)	Appends the specified string to the builder.
insert(int offset, String s)	Inserts the string at the specified offset.
replace(int start, int end, String s)	Replaces characters between start and end with the given string.
delete(int start, int end)	Deletes characters between start and end.
reverse()	Reverses the sequence of characters.
capacity()	Returns the current capacity of the builder.
ensureCapacity(int min)	Ensures a minimum capacity for the builder.
setLength(int newLength)	Sets the length of the sequence.
charAt(int index)	Returns the character at the specified index.
setCharAt(int index, char c)	Replace character at given index
toString()	Converts the builder to a String object.

```
public class StringBuilderExample {
    public static void main(String[] args) {
        //Declaration
        //StringBuilder sb1 = "Hello";  //gives error
        StringBuilder sb1 = new StringBuilder("Hello ");

        //Append
        //sb1+= "World";   //Gives error
        sb1.append("World");
        System.out.println(sb1 + " " +String.valueOf(sb1.length()));
        // Hello World 11

        //replace at given index ( can't do with String)
        sb1.setCharAt(2, 'f');
        System.out.println(sb1);
        //Heflo World

        //Add new string
        sb1.insert(6, "My ");
        System.out.println(sb1);
        //Heflo My World

        //Set Length
        sb1.setLength(5);
        System.out.println(sb1);
        //Heflo
    }
}
```

7. Packages

A package in Java is a namespace that organizes classes and interfaces, making them easier to manage and avoid naming conflicts. Packages act like folders in a file directory, grouping related classes and interfaces together.

- Built-in Packages: Provided by Java libraries.
  Example: java.util, java.io, java.net
- User-defined Packages: Created by the developer to group their own classes.

Naming Convention: -

To avoid conflicts with packages from other developers, Java uses a reverse domain name convention.

Guidelines:

- Start with a domain name: Use your organization or project's domain name in reverse.
  Example: com.example.projectname
- Lowercase Letters: Use lowercase letters for package names.
  Example: com.example.utils
- Subpackages: Use dot-separated names to create subpackages for further organization.
  Example: com.example.utils.fileio

Defining a package: -

```
package mypackage;

public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass!");
    }
}
```

Importing a package: -

1. Importing specific class
   `import mypackage.MyClass;`

2. Importing all classes in package
   `import mypackage.*;`

Some built-in packages:-

Package	Description
`java.lang`	Core classes (e.g., `String`, `Math`)
`java.util`	Utility classes (e.g., `ArrayList`)
`java.io`	Input and output (e.g., `File`, `BufferedReader`)
`java.net`	Networking (e.g., `Socket`, `URL`)
`java.sql`	Database access (e.g., `Connection`)

Notes: -

- File Structure: The directory structure must match the package name.
  For package com.example.utils;, the file should be in the directory: com/example/utils/
- java.lang Package: This is imported by default, so you don't need to explicitly import it.
- Default Package: Classes without a package declaration are in the default package but are discouraged for larger projects.

8. Wrapper classes

Wrapper classes in Java are part of the java.lang package and provide a way to use primitive data types (int, char, boolean, etc.) as objects. Each primitive type has a corresponding wrapper class.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Why Use Wrapper Classes?

- Collections Framework Compatibility: Collections like ArrayList, HashMap, etc., work with objects, not primitives.
- Utilities and Methods: 1
- Nullability: Unlike primitives, wrapper class objects can be null.
- Serialization: Wrappers can be serialized, whereas primitives cannot.
- Reflection: Wrappers are used in reflection as they are objects.

Memory Storage

Primitive Types:

- Stored in the stack memory.
- Lightweight and directly store values.

Wrapper Classes:

- Objects are stored in the heap memory.
- Require additional memory due to object overhead.

Auto-Boxing and Unboxing

Auto-Boxing: Automatically converts a primitive type to its corresponding wrapper class.

Unboxing: Automatically converts a wrapper class object back to its corresponding primitive type.

```java
public class LearnWrapper {
    public static void main(String[] args) {

        int a = 32;
        Integer aba = a;  //AutoBoxing

        int uba = aba;    //Unboxing

        System.out.println(aba+" " + uba);  //32 32
    }
}
```

Common methods: -

Wrapper Class	Method	Description
Integer	parseInt(String s)	Converts a String to an int.
Double	parseDouble(String s)	Converts a String to a double.
Boolean	parseBoolean(String s)	Converts a String to a boolean.
Integer	valueOf(String s)	Returns an Integer object holding the value.
Integer	toString(int i)	Converts an int to a String.

```java
import java.util.ArrayList;
import java.util.Collections;

public class LearnWrapper {
    public static void main(String[] args) {
        //int a =null;    is a exception
        Integer a = null;
        System.out.println(a);   //null

        Integer b = Integer.parseInt("23");
        System.out.println(b);  //23

        String c = Integer.toString(23);
        System.out.println(c + " " + c.getClass().getName());
        //23 java.lang.String

        //Only Wrappers can be used with collection
        //ArrayList<int> d = new ArrayList<>();   is Error
        ArrayList<Integer> d = new ArrayList<>();
    }
}
```

9. Var keyword

Introduced in Java 10, the var keyword allows for **local variable type inference**(LVTI), enabling developers to omit explicitly specifying the type of the variable. The compiler determines the type based on the value assigned.

1. **Scope:** var can only be used for local variables (inside methods, for loops, etc.), not for fields or method parameters.
2. **Initialization Required:** The variable must be initialized during declaration because the compiler infers its type from the assigned value.
3. **Type-Safe:** Although the type is inferred, Java remains strongly typed. The variable's type is determined at compile time and cannot change.

Example with primitive type

```
public class LearnVar {
    //var k = 10;       This is error(IT IS NOT LOCAL VARIABLE)
    public static void main(String[] args) {
        var message = "Hello, Java!";  // Inferred as String
        var number = 10;               // Inferred as int
        var price = 99.99;             // Inferred as double
        //var i;            //This is error, var variables must be initialized
        System.out.println("Message: " + message);
        System.out.println("Number: " + number);
        System.out.println("Price: " + price);
    }
}
```

In loop

```
public class LearnVar {
    public static void main(String[] args) {
        var names = List.of("Alice", "Bob", "Charlie");

        for (var name : names) {  // Inferred as String
            System.out.println(name);
        }
    }
}
```

with collection

```
public class LearnVar {
    public static void main(String[] args) {
        var map = new HashMap<String, Integer>();  // Inferred as
HashMap<String, Integer>
        map.put("Java", 10);
        map.put("Python", 8);

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

Rules for using var: -

1. Cannot be used without initialization
2. Cannot change the type later
3. Can’t assign null value
4. Can’t be used as method parameters of fields

10. Interview Questions

1. What Is difference between compile time and runtime?

Compile time: - refers to period during which source code is translated into machine code or byte code by the compiler.

Runtime: - also called execution time, refers to the period during which a program is running and performing its operation. Compiled code is executed by processor or virtual machine. The program interacts with system resources perform computation and produces desired output.

Aspect	Compile-Time	Runtime
When It Happens	During the compilation of code into an executable format.	During the actual execution of the program.
Purpose	Checks for syntax, type errors, and converts code to machine-readable format.	Executes the logic of the program.
Error Detection	Detects syntax, type, and semantic errors.	Detects dynamic issues like divide-by-zero, null pointers, or out-of-bounds errors.
Performance Impact	No impact on program performance; happens before execution.	Directly impacts performance during execution.
Examples of Actions	- Syntax checking - Type checking - Optimization of code.	- Memory allocation - Input/output - Loops and function calls.

2. What are strictly typed and loosely typed languages?

Strictly typed: - A language is strictly typed if it enforces strict rules about how variables are used and assigned, typically ensuring that data types are explicitly declared and consistently maintained.

Type mismatches are not allowed. For example, you cannot assign a string to an integer variable without explicit conversion. Languages like Java, C, python, C++.

```
public class StrictTypingExample {
    public static void main(String[] args) {
        int number = 10; // Integer type variable
        // number = "Hello"; // This will cause a compile-time error
    }
}
```

Note: - Python is dynamically strictly typed, variable type is confirmed at runtime, but afterwards it is strict

Code example:

```
1  a = True
2  a = 3   #this is allowed, but now a is integer
3  a += "Hiii"
4  print(a)
```

```
Traceback (most recent call last):
  File "/home/main.py", line 3, in <module>
    a += "Hiii"
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Loosely typed: - A language is loosely typed if it allows more flexibility in handling types, often performing automatic type conversion (type coercion) when needed. This can lead to fewer type-related errors but might introduce unexpected behavior. Languages like JavaScript, PHP

```
//JavaScript code
console.log(3 - "2");     //return 1
console.log(3 + "2");     //return 32
console.log(3 * "ABC")    // return NAN (But no error)
```

3. Is java interpreted or compiled language?

Interpreter:

- Line-by-line execution: Interpreters translate and execute code one line at a time.
- Slower execution: This can lead to slower performance compared to compiled languages.
- Platform-independent: Interpreters can run on different platforms as long as the interpreter is available.
- Easier debugging: Errors are identified and fixed line by line.
- Examples: Python, Ruby, JavaScript, PHP

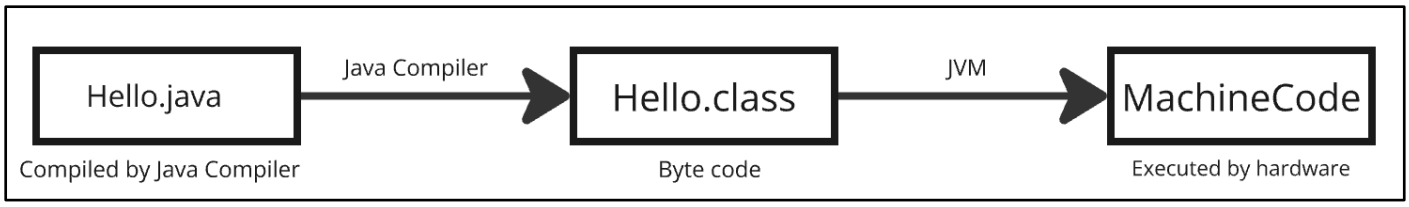
Compiler:

- Complete translation: Compilers translate the entire code into machine code before execution.
- Faster execution: Compiled code executes directly on the machine, resulting in faster performance.
- Platform-dependent: Compiled code is specific to a particular platform.
- More difficult debugging: Errors are identified after the entire code is compiled.
- Examples: C, C++, C#

Java: A Hybrid Approach

Compilation: Java source code is first compiled into bytecode by the Java compiler. Bytecode is a platform-independent intermediate code.

Interpretation: The Java Virtual Machine (JVM) interprets the bytecode at runtime.



This two-step process offers the best of both worlds:

- Platform independence: Bytecode can run on any system with a JVM.
- Performance optimization: The JVM can optimize the bytecode for the specific hardware and operating system.

4. Why Are Strings Immutable?

- Security: Prevents accidental or malicious modification of data (e.g., passwords).
- Thread Safety: Immutable strings can be shared between threads without synchronization issues.
- Memory Efficiency: Reusing strings in the pool saves memory.

5. Can var is used only for local variable?

1. Scope Clarity: Local variables have a small, limited scope, making the inferred type easy to understand and track, unlike fields or method parameters, which have broader scopes.
2. Initialization Requirement: var requires immediate initialization for type inference, which is impractical for fields or method parameters that may not have initial values.
3. API Clarity: Explicit types are necessary for fields and method signatures to make a class's API clear and self-documenting, avoiding ambiguity in public interfaces.
4. Design Philosophy: Java prioritizes strong typing and maintainability, and restricting var to local variables prevents potential pitfalls and complexity in understanding code structure.

6. Can var type can be returned from function?

No, you cannot directly return var from a method in Java because var is not a data type. It is just a syntax feature used for local variable type inference. The actual type of the variable inferred by var must be explicitly declared when returning it from a method.

```
public var getValue() {  // Compilation Error
    var value = 10;  // Inferred as int
    return value;
}

public int getIntValue(){    //this will work
    var value = 10;
    return value;
}
```

But in case we really don't know output type, we have 2 ways: use Object as output type, or use generics

```
public Object getValue() {  // Method 1
    var value = 10;  // Inferred as int
    return value;
}

public <T> T getIntValue(T value){    //Method 2 ( use Generics)
    return value;
}
```

- var is not a type; it is a shorthand for local variable declarations, so it cannot be used as a method's return type.
- Methods must have explicit return types, even if var is used inside the method.
- Use Object or generics if the type cannot be determined beforehand.