B-TECH JAVA-PROGRAMMING (UNIT-1)

1. **Introduction to Java:** Java is a popular programming language used to build different types of applications, from mobile apps to large-scale enterprise systems.

2. History of Java:

- **1991**: Java began as *Oak* for consumer electronics at Sun Microsystems.
- **1995**: Renamed *Java*, it became a web-focused, platform-independent language.
- **1996**: Java 1.0 released with "write once, run anywhere" capability.
- **2009**: Oracle acquired Java with Sun Microsystems.

Java is now a key language for web, enterprise, and mobile apps.

3. Features of Java:

Here are the key features of Java:

1. **Object-Oriented**: Java uses the object-oriented programming model, making it easy to structure code using classes and objects.
2. **Platform-Independent**: Java's "write once, run anywhere" principle is powered by the Java Virtual Machine (JVM), allowing Java code to run on any platform without modification.
3. **Simple**: Java is designed to be easy to learn with a clean syntax, making it accessible for beginners.
4. **Secure**: Java offers built-in security features such as bytecode verification, security managers, and sandboxing.
5. **Robust**: Java has strong memory management, automatic garbage collection, and exception handling to create stable applications.
6. **Multithreaded**: Java natively supports multithreading, allowing multiple tasks to run concurrently.
7. **Portable**: Since Java is platform-independent at both the source and binary levels, it's highly portable.
8. **High Performance**: With features like Just-In-Time (JIT) compilation, Java offers good performance for most applications.
9. **Distributed**: Java has networking capabilities to build distributed applications through its APIs like RMI and CORBA.
10. **Dynamic**: Java supports dynamic loading of classes, linking in new class libraries, methods, and objects during runtime.

4. C++ vs. Java:

1. Platform Dependency

- **C++**: Platform-dependent; compiled into machine code specific to an operating system.
- **Java**: Platform-independent; runs on any system with a JVM ("write once, run anywhere").

2. Memory Management

- **C++**: Manual memory management using new and delete. The programmer is responsible for freeing memory.
- **Java**: Automatic garbage collection; the JVM handles memory management.

3. Object-Oriented Paradigm

- **C++**: Supports both procedural and object-oriented programming (multi-paradigm).
- **Java**: Purely object-oriented (everything is part of a class, except for primitive types).

4. Pointers

- **C++**: Allows direct use of pointers, providing greater control over memory but increasing the risk of memory errors.
- **Java**: No explicit pointer usage; references are used instead for greater safety.

5. Multiple Inheritance

- **C++**: Supports multiple inheritance (a class can inherit from more than one class).
- **Java**: Does not support multiple inheritance with classes, but it allows multiple inheritance through interfaces.

6. Compilation and Interpretation

- **C++**: Compiled directly to machine code.
- **Java**: Compiled to bytecode, which is then interpreted or compiled by the JVM.

7. Exception Handling

- **C++**: Exception handling is supported, but it's optional and less integrated.
- **Java**: Strong emphasis on exception handling, and it is integrated into the language (checked exceptions).

8. Speed and Performance

- **C++**: Typically faster due to direct compilation into machine code.
- **Java**: Slightly slower because of JVM's bytecode interpretation, but Just-In-Time (JIT) compilation improves performance.

5. Java Data Types:

Java has two main categories of data types: **Primitive Data Types** and **Non Primitive Data Types**.

1. Primitive Data Types

These are the basic data types in Java that store simple values.

Examples:

- **byte**: Stores small integers, 1 byte (8 bits).

  java
  Copy code
  ```
  byte age = 25;  // Example: age is 25
  ```

- **short**: Stores medium-sized integers, 2 bytes (16 bits).

  java
  Copy code
  ```
  short year = 2024;  // Example: year is 2024
  ```

- **int**: Stores large integers, 4 bytes (32 bits).

  java
  Copy code
  ```
  int population = 1000000;  // Example: population is 1 million
  ```

- **long**: Stores very large integers, 8 bytes (64 bits).

  java
  Copy code
  ```
  long distance = 15000000000L;  // Example: distance is 15 billion (note the 'L' suffix)
  ```

- **float**: Stores decimal numbers, 4 bytes (32 bits). Use for single-precision floating-point numbers.

  java
  Copy code
  ```
  float price = 19.99f;  // Example: price is 19.99 (note the 'f' suffix)
  ```

- **double**: Stores larger decimal numbers, 8 bytes (64 bits). Use for double-precision floating-point numbers.

  java
  Copy code
  ```
  double temperature = 98.6;  // Example: temperature is 98.6
  ```

- **char**: Stores a single character or Unicode value, 2 bytes (16 bits).

  java
  Copy code
  ```
  char grade = 'A';  // Example: grade is 'A'
  ```

- **boolean**: Stores true or false values, 1 bit (though its exact size is JVM-dependent).

  java
  Copy code
  ```
  boolean isJavaFun = true;  // Example: true or false values
  ```

2. Non Primitive **Data Types**

These data types store references (addresses) to objects in memory. Common reference types include classes, arrays, and interfaces.

Examples:

- **String**: A sequence of characters, stored as an object.

  ```
  String name = "Java Programming";  // Example: "Java Programming"
  ```

- **Array**: A collection of data of the same type.

  ```
  int[] numbers = {1, 2, 3, 4, 5};  // Example: Array of integers
  ```

- **Class Objects**: Stores references to objects created from classes.

  ```
  class Car {
     String model;
  }

  Car myCar = new Car();  // Example: Object of Car class
  myCar.model = "Tesla";  // Setting the model field of the car object
  ```

Summary

- **Primitive data types** (byte, short, int, long, float, double, char, boolean) hold actual values.
- **NON Primitive data types** (like String, arrays, and objects) hold references to memory locations where the actual data is stored.

6. Tokens:

In Java, **tokens** are the smallest elements in a program that have some meaning. A Java program is made up of various tokens. The following are the key types of tokens in Java:

- ### Keywords

These are reserved words in Java that have a specific meaning and cannot be used as identifiers (e.g., class names, variable names).

Examples:

```
public class Example {
    public static void main(String[] args) {
        int number = 10;  // "int", "public", "class", "static", and "void" are keywords
    }
}
```

- ### Identifiers

Identifiers are names given to variables, methods, classes, etc., created by the programmer.

Examples:

```
int age = 25;  // "age" is an identifier
String name = "John";  // "name" is an identifier
```

- ### Literals

Literals are constant values that appear directly in the program. They can be numeric, character, string, or boolean values.

Examples:

```
int number = 100;       // 100 is an integer literal
char grade = 'A';       // 'A' is a character literal
boolean isTrue = true;  // true is a boolean literal
String message = "Hello, World!";  // "Hello, World!" is a string literal
```

- ### Operators

Operators perform operations on variables and values. Java supports various types of operators like arithmetic, relational, logical, bitwise, etc.

Examples:

```
int a = 10, b = 20;

int sum = a + b;  // "+" is an arithmetic operator
boolean result = a < b;  // "<" is a relational operator
```

- ### Separators (Delimiters)

Separators are used to define the structure of a program. Common separators include `;`, `()`, `{}`, `[]`, and `,`.

Examples:

```
public class Example {   // "{" and "}" are block separators
    public static void main(String[] args) {   // "(" and ")" are parentheses separators
        int[] numbers = {1, 2, 3};  // "{}" separates the array elements, "," separates the elements
    }
}
```

- ### Comments

Comments are used to make the code more readable and are ignored by the compiler. Java supports single-line (`//`) and multi-line comments (`/* */`).

Examples:

```
// This is a single-line comment

/*
```

```
  This is a multi-line comment
*/
```

Example Combining Different Tokens:

```
public class Example {          // "public", "class" are keywords, "Example" is an identifier
    public static void main(String[] args) {  // "String", "args" are identifiers, "[]" is a separator
        int number = 10;        // "int" is a keyword, "number" is an identifier, "10" is a literal
        System.out.println("Number: " + number);  // "println" is an identifier, "+" is an operator, ";" is a separator
    }
}
```

7. Java Compiler and Interpreter: In Java, the terms **compiler** and **interpreter** refer to different components of the Java execution process. Here's how they work together:

- **Java Compiler :**

- **Role**: The Java compiler translates Java source code (.java files) into bytecode (.class files).
- **Function**: It performs syntax checking, type checking, and generates intermediate code (bytecode) that is platform-independent.
- **Process**:
    1. **Compilation**: The source code written in Java is compiled by the javac compiler into bytecode, which is a set of instructions for the Java Virtual Machine (JVM).
    2. **Output**: The output is a .class file containing bytecode, which can be executed by the JVM.

- **Java Interpreter (JVM)**

- **Role**: The Java Virtual Machine (JVM) interprets and executes the bytecode generated by the Java compiler.
- **Function**: The JVM converts bytecode into machine code and executes it. It can also optimize code execution through Just-In-Time (JIT) compilation, which compiles bytecode to machine code at runtime for better performance.
- **Process**:
    1. **Loading**: The JVM loads the compiled bytecode from .class files.
    2. **Execution**: It interprets the bytecode or uses JIT compilation to convert it to native machine code for execution.
    3. **Runtime Environment**: It provides a runtime environment including memory management (garbage collection), security checks, and other services.

Summary:

- **Java Compiler** converts Java source code into platform-independent bytecode.
- **Java Interpreter** (part of the JVM) executes the bytecode on any platform with a JVM, making Java platform-independent.

By using both the compiler and interpreter (or the JVM with JIT), Java achieves its goal of "write once, run anywhere."

8.Java Bytecode:

 Definition: Bytecode is a set of instructions generated by the Java compiler from Java source code. It is not machine code but an intermediate code that can be executed by the Java Virtual Machine (JVM).

 Format: Bytecode is a binary representation of the Java source code. It is stored in .class files after compilation.

9. Java Development Kit (JDK):

The **Java Development Kit (JDK)** is a comprehensive software development package used for developing Java applications. It includes tools, libraries, and other components necessary for building, compiling, and running Java programs. Here's a detailed overview:

Key Components of the JDK

1. **Java Compiler (**
   - **Function**: Converts Java source code (.java files) into Java bytecode (.class files).
   - **Command**: javac Example.java
2. **Java Virtual Machine**
   - **Function**: Executes Java bytecode on the JVM. It interprets or compiles bytecode into native machine code.
   - **Command**: java Example
3. **Java Runtime Environment (JRE)**
   - **Function**: Provides the runtime environment to run Java applications. It includes the JVM and core libraries but does not include development tools.
   - **Note**: The JRE is included in the JDK, but it is a separate package that can be installed independently for running Java applications.
4. **Java API (Application Programming Interface)**
   - **Function**: A collection of pre-written classes and interfaces (standard libraries) that provide functionality such as data structures, networking, I/O operations, and more.
   - **Examples**: java.lang, java.util, java.io

11. Naming Conventions: These are rules and guidelines for naming variables, classes, methods, etc., in Java to keep the code organized and readable.

12. Applications of Java: Java is used in various applications like

- web applications
- mobile apps
- enterprise systems

- games, and more.

13. Using Decision Making Constructs:

- *if, if-else*: These are used to make decisions in a program. For example, if a condition is true, do something; else do something else.

- *if-else if, switch statement*: These are also used for making decisions but allow multiple conditions to be checked.

1. `if` Statement

The `if` statement allows the program to execute a block of code only if a specified condition is true.

```
if (condition) {
   // Code to be executed if condition is true
}
```

Example:

```
int x = 10;
if (x > 5) {
   System.out.println("x is greater than 5");
}
```

2. `if-else` Statement

The `if-else` statement provides an alternative block of code to execute if the condition is false.

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

Example:

```
int x = 4;
if (x > 5) {
   System.out.println("x is greater than 5");
} else {
   System.out.println("x is not greater than 5");
}
```

3. `if-else if-else` Statement

This construct allows for multiple conditions to be checked in sequence. Once a condition is met, the corresponding block of code is executed.

```
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition2 is true
} else {
    // Code to be executed if none of the conditions are true
}
```

Example:

```
int x = 8;
if (x > 10) {
    System.out.println("x is greater than 10");
} else if (x > 5) {
    System.out.println("x is greater than 5 but less than or equal to 10");
} else {
    System.out.println("x is 5 or less");
}
```

4. `switch` Statement

The `switch` statement is used when a variable is compared against multiple possible values, and a specific block of code is executed based on the value.

```
switch (expression) {
    case value1:
        // Code to be executed if expression == value1
        break;
    case value2:
        // Code to be executed if expression == value2
        break;
    // more cases
    default:
        // Code to be executed if no case matches
}
```

Example:

```
int day = 2;
switch (day) {
    case 1:
        System.out.println("Sunday");
        break;
    case 2:
        System.out.println("Monday");
        break;
    case 3:
        System.out.println("Tuesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

14. Iterative Constructs:

- **while loop:** Repeats a block of code as long as a condition is true.
- **for loop:** Repeats a block of code a certain number of times.
- **do-while loop:** Similar to a while loop but checks the condition after running the loop once.
- **break:** Exits a loop early.
- **continue:** Skips the rest of the loop and continues with the next iteration.

1. for Loop

The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and update expression, which are all included in the loop declaration.

```
for (initialization; condition; update) {
    // Code to be executed
}
```

Example

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

In this example, the loop will execute five times, printing the values from 0 to 4.

2. Enhanced for Loop (For-each Loop)

The enhanced for loop, or for-each loop, is used for iterating over arrays or collections like ArrayList. It provides a simple way to traverse through elements without dealing with the index or iterator.

```
for (type element : array) {
    // Code to be executed
}
```

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}
```

3. while Loop

The while loop repeats a block of code as long as the specified condition remains true. The condition is checked before entering the loop, making it a pre-condition loop.

```
while (condition) {
   // Code to be executed
}
```

Example

```
int i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
```

In this example, the loop will continue to execute as long as i is less than 5.

4. do-while Loop

The do-while loop is similar to the while loop, except that the condition is checked **after** the code block is executed. This guarantees that the loop will execute at least once, regardless of the condition.

```
do {
   // Code to be executed
} while (condition);
```

Example:

```
int i = 0;
do {
   System.out.println(i);
   i++;
} while (i < 5);
```

In this case, the loop executes and checks the condition at the end.

5. break and continue Statements in Loops

- **break**: Exits the loop immediately, even if the loop condition is still true.
- **continue**: Skips the current iteration and proceeds to the next iteration of the loop.

Example with break:

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break;  // Stops the loop when i is 5
   }
   System.out.println(i);
}
```

Example with continue:

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;  // Skips the rest of the loop when i is 3
    }
    System.out.println(i);
}
```

15. <u>Defining Arrays:</u>

Arrays are used to store multiple values in a single variable. For example, an array can hold multiple integers.

Key Features of Arrays in Java:

1. **Fixed Size**: The size of an array is determined when it is created and cannot be changed later.
2. **Zero-based Indexing**: Array indices start at 0, so the first element is accessed using index 0.
3. **Homogeneous Elements**: All elements in an array must be of the same data type.
4. **Random Access**: You can access any element in the array directly using its index.

Declaring and Creating Arrays

Arrays are declared by specifying the data type of the elements, followed by square brackets (`[]`), and then the array name. You can either initialize the array immediately or later in the code.

Syntax:

```
dataType[] arrayName;
```

or

```
dataType arrayName[];
```

Example:

```
int[] numbers;  // Declares an array of integers
```

Creating an Array

To create an array, you use the `new` keyword and specify the size of the array (number of elements).

Syntax:

```
arrayName = new dataType[arraySize];
```

Example:
```
numbers = new int[5];  // Creates an array that can hold 5 integers
```

Initializing an Array

You can also declare, create, and initialize an array in one line.

Syntax:
```
dataType[] arrayName = new dataType[arraySize];
```

Example:
```
int[] numbers = new int[5];  // Declares and creates an array of 5 integers
```

You can also initialize the array with values directly at the time of declaration:

```
int[] numbers = {1, 2, 3, 4, 5};  // Array with 5 elements initialized
```

Accessing Array Elements

Array elements are accessed using their index, which starts at 0. To access an element, use the array name followed by the index in square brackets.

Example:
```
int[] numbers = {10, 20, 30, 40, 50};
System.out.println(numbers[0]);  // Outputs 10
System.out.println(numbers[3]);  // Outputs 40
```

Modifying Array Elements

You can change the value of an array element by assigning a new value to it using its index.

Example:
```
int[] numbers = {1, 2, 3, 4, 5};
numbers[2] = 10;  // Changes the third element (index 2) to 10
System.out.println(numbers[2]);  // Outputs 10
```
16. Types of Array: Java supports different types of arrays like single-dimensional arrays and multi-dimensional arrays (like a matrix).

1. Single-Dimensional Array

A single-dimensional array is the simplest form of an array in Java. It stores a collection of elements in a linear format, where each element is accessed by a single index.

Declaration:
```
dataType[] arrayName = new dataType[size];
```

Example:
```
int[] numbers = new int[5];  // Declares an integer array with 5 elements
numbers[0] = 10;  // Assigns the value 10 to the first element
numbers[1] = 20;
// Alternatively, you can initialize directly:
int[] numbers = {10, 20, 30, 40, 50};
```

Accessing elements:

```
System.out.println(numbers[0]);  // Outputs: 10
```

Use case:

Single-dimensional arrays are useful for storing and working with a simple list of elements, like a list of marks, temperatures, or any sequential data.

---

2. Multidimensional Arrays

Multidimensional arrays are arrays of arrays, where each element itself can be an array. The most common type of multidimensional array is the **two-dimensional (2D) array**, which is often used to represent data in matrix or table format.

Types of Multidimensional Arrays:

a) Two-Dimensional (2D) Array

A two-dimensional array represents a table with rows and columns. Each element is accessed using two indices, one for the row and one for the column.

Declaration:

```
dataType[][] arrayName = new dataType[rows][columns];
```

Example:

```
int[][] matrix = new int[3][3];  // 2D array with 3 rows and 3 columns
matrix[0][0] = 1;  // First element in the first row

// Alternatively, initialize directly:
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Accessing elements:

```
System.out.println(matrix[1][2]);  // Outputs: 6 (element at row 1, column 2)
```

Use case:

2D arrays are often used for storing and manipulating matrices, tables, and grids, like a chessboard, school gradebooks, or game boards.

---

b) Three-Dimensional (3D) Array

A three-dimensional array represents data in a cubic structure, and each element is accessed using three indices.

Declaration:

```
dataType[][][] arrayName = new dataType[size1][size2][size3];
```

Example:

```
int[][][] cube = new int[3][3][3];  // 3D array with 3 planes, 3 rows, and 3 columns
cube[0][0][0] = 1;

// Alternatively, initialize directly:
int[][][] cube = {
    {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    },
    {
        {10, 11, 12},
        {13, 14, 15},
        {16, 17, 18}
    },
    {
        {19, 20, 21},
        {22, 23, 24},
        {25, 26, 27}
    }
};
```

Accessing elements:

```
System.out.println(cube[1][2][0]);  // Outputs: 16 (second plane, third row
```

17. <u>Operation on String:</u> Strings in Java are sequences of characters. This topic covers how to manipulate strings, like concatenation (joining strings), finding length, etc.

1. Creating Strings

You can create a string in Java in two main ways:

- Using string literals.
- Using the new keyword.

Example:

```
String s1 = "Hello";  // String literal
String s2 = new String("World");  // Using new keyword
```

2. Concatenation

You can concatenate strings using the + operator or the concat() method.

Example:

```
String s1 = "Hello";
String s2 = "World";
String result = s1 + " " + s2;  // Using + operator
System.out.println(result);  // Outputs: "Hello World"

String result2 = s1.concat(" ").concat(s2);  // Using concat() method
```

```
System.out.println(result2);  // Outputs: "Hello World"
```

3. Length of String

The length() method returns the number of characters in the string.

Example:

```
String s = "Hello World";
int len = s.length();
System.out.println(len);  // Outputs: 11
```

4. Accessing Characters

The charAt() method returns the character at a specific index in the string. Remember, string indices are 0-based.

Example:

```
String s = "Hello";
char ch = s.charAt(1);
System.out.println(ch);  // Outputs: 'e'
```

5. Substring

The substring() method extracts a part of the string. You can provide the start index and (optionally) the end index.

Syntax:

```
String substring(int beginIndex);
String substring(int beginIndex, int endIndex);
```

Example:

```
String s = "Hello World";
String sub1 = s.substring(6);  // Extracts substring from index 6 to end
System.out.println(sub1);  // Outputs: "World"

String sub2 = s.substring(0, 5);  // Extracts substring from index 0 to 5 (exclusive)
System.out.println(sub2);  // Outputs: "Hello"
```

6. String Comparison

You can compare strings using the following methods:

- **equals()**: Compares two strings for equality, considering case.
- **equalsIgnoreCase()**: Compares two strings for equality, ignoring case.
- **compareTo()**: Compares two strings lexicographically (in dictionary order).

Example:

```
String s1 = "Hello";
String s2 = "hello";
System.out.println(s1.equals(s2));  // Outputs: false
System.out.println(s1.equalsIgnoreCase(s2));  // Outputs: true

String s3 = "Apple";
String s4 = "Banana";
System.out.println(s3.compareTo(s4));  // Outputs: -1 (since "Apple" comes before "Banana")
```

7. Changing Case

Java provides methods to convert strings to lowercase or uppercase:

- **toLowerCase()**: Converts all characters to lowercase.
- **toUpperCase()**: Converts all characters to uppercase.

Example:

```
String s = "Hello World";
System.out.println(s.toLowerCase());  // Outputs: "hello world"
System.out.println(s.toUpperCase());  // Outputs: "HELLO WORLD"
```

8. Trim

The trim() method removes leading and trailing whitespace from the string.

Example:

```
String s = "  Hello World  ";
System.out.println(s.trim());  // Outputs: "Hello World"
```

9. Replace Characters or Substrings

The replace() method replaces occurrences of a character or a substring with another character or substring.

Example:

```
String s = "Hello World";
String replaced = s.replace('l', 'x');  // Replaces all 'l' with 'x'
System.out.println(replaced);  // Outputs: "Hexxo Worxd"

String replacedSub = s.replace("World", "Java");  // Replaces "World" with "Java"
System.out.println(replacedSub);  // Outputs: "Hello Java"
```

10. Split a String

The split() method splits a string into an array of substrings based on a delimiter.

Example:

```
String s = "apple,orange,banana";
String[] fruits = s.split(",");
for (String fruit : fruits) {
    System.out.println(fruit);
}
// Outputs:
// apple
// orange
// banana
```

11. String Contains

The contains() method checks if a string contains a particular sequence of characters.

Example:

```
String s = "Hello World";
boolean contains = s.contains("World");
System.out.println(contains);  // Outputs: true
```

12. Index of a Character or Substring

The indexOf() method returns the index of the first occurrence of a character or substring. It returns -1 if the character or substring is not found.

Example:

```
String s = "Hello World";
int index = s.indexOf('W');
System.out.println(index);  // Outputs: 6
```

13. StartsWith and EndsWith

- **startsWith()**: Checks if a string starts with a specified prefix.
- **endsWith()**: Checks if a string ends with a specified suffix.

Example:

```
String s = "Hello World";
System.out.println(s.startsWith("Hello"));  // Outputs: true
System.out.println(s.endsWith("World"));  // Outputs: true
```

14. Join Strings

The join() method joins multiple strings using a delimiter.

Example:

```
String joined = String.join(", ", "apple", "orange", "banana");
System.out.println(joined);  // Outputs: "apple, orange, banana"
```

15. Convert String to Character Array

The toCharArray() method converts a string into a character array.

Example:

```
String s = "Hello";
char[] charArray = s.toCharArray();
for (char ch : charArray) {
    System.out.println(ch);
}
// Outputs:
// H
// e
// l
// l
// o
```

16. String to Primitive Types (Parse)

You can convert strings to primitive types using methods like Integer.parseInt(), Double.parseDouble(), etc.

Example:

```
String s = "123";
int num = Integer.parseInt(s);
System.out.println(num);  // Outputs: 123
```

Summary of Operations on Strings:

- **Creation**: String literals or new keyword.
- **Concatenation**: Using + or concat().
- **Comparison**: Using equals(), equalsIgnoreCase(), or compareTo().
- **Substring extraction**: Using substring().
- **Modification**: Using replace(), toUpperCase(), toLowerCase().
- **Search**: Using indexOf(), contains().
- **Splitting and Joining**: Using split() and join().
- **Conversion**: Converting strings to primitive types and vice versa.

These operations allow Java developers to work effectively with strings in various ways, including manipulation, searching, and transforming them for specific purposes.

18. <u>String Handling:</u> This includes all the operations you can perform on strings, like comparison, searching, replacing characters, etc.

String handling in Java refers to the various techniques and methods used to create, manipulate, and manage strings. Strings are widely used in Java, and Java provides a rich set of features and methods for string manipulation. Strings in Java are **immutable**, meaning once a string is created, its value cannot be changed. This immutability provides efficiency and security, but Java offers a variety of tools to work with strings flexibly.