Q1. What are the Conditional Operators in Java?

Conditional Operator in Java

In Java, **conditional operators** check the condition and decides the desired result on the basis of both conditions. In this section, we will discuss the **conditional operator in Java.**

Types of Conditional Operator

There are three types of the conditional operator in Java:

- Conditional AND
- Conditional OR
- Ternary Operator

Operator	**Symbol**
Conditional or Logical AND	&&
Conditional or Logical OR	||
Ternary Operator	?:

Conditional AND

The operator is applied between two Boolean expressions. It is denoted by the two AND operators (&&). It returns true if and only if both expressions are true, else returns false.

Expression1	**Expression2**	**Expression1 && Expression2**
True	False	False
False	True	False
False	False	False
True	True	True

Conditional OR

The operator is applied between two Boolean expressions. It is denoted by the two OR operator (||). It returns true if any of the expression is true, else returns false

Expression1	Expression2	Expression1 || Expression2
True	True	True
True	False	True
False	True	True
False	False	False

Q2. What are the types of operators based on the number of operands?

Java Operators

An operator is a **symbol that performs a specific operation on one, two, or three operands, producing** a result. The type of the operator and its operands determine the kind of operation performed on the operands and the type of result produced.

Operators in Java can be categorized based on two criteria:

- **Number of operands** – There are three types of operators based on the number of operands. An operator is called a unary, binary, or ternary operator based on the number of operands. If an operator takes one operand, it is called a unary operator; if it takes two operands, it is called a binary operator; if it takes three operands, it is called a **ternary operator**.
- **Type of operation they perform** – An operator is called an **arithmetic operator**, a **relational operator**, a **logical operator**, or a **bitwise operator**, depending on the kind of operation it performs on its operands.

Operator	Description
+ – * /	Arithmetic unary or binary operators
++ —	Increment and decrement unary operators

Operator	Description
== !=	Equality operators
< > <= =>	Relational operators
! & |	Logical operators
&& || ?:	Conditional operators
= += -= *= /= %=	Assignment operators
&= |= ^= <<= >>= >>>=	Assignment operators
& | ~ ^ << >> >>>	Bitwise operators
-> ::	Arrow and method reference operators
new	Instance creation iterator
instanceOf	Type comparison operator
+	String concatenation operator

Let us learn about few most used operators with examples.

2. Assignment Operator

- An assignment operator (=) is used to assign a value to a variable.
- It is a binary operator. It takes two operands.
- The value of the right-hand operand is assigned to the left-hand operand.
- The left-hand operand must be a variable.

```
//26 is the right-hand operand.
//counter is the left-hand operand, which is a variable of type int.

int counter = 26;
```

Java ensures that the value of the right-hand operand of the assignment operator is assignment compatible to the data type of the left-hand operand. Otherwise, a **compile-time error** occurs.

In case of reference variables, you may be able to compile the source code and get a runtime ClassCastException error if the object represented by the right-hand operand is not assignment compatible to the reference variable as the left-hand operand.

3. Arithmetic Operators

- Operators like (**+** (plus), **–** (minus), ***** (multiply), **/** (divide)) are called arithmetic operators in Java.
- It can only be used with numeric type operands. It means, both operands to arithmetic operators must be one of types byte, short, char, int, long, float, and double.
- These operators cannot have operands of boolean primitive type and reference type.

```
int sum = 10 + 20;



int difference = 50 - 20;



long area = 20l * 30l;



int percentage = 20 / 100;
```

3.1. Unary Arithmetic Operators

Operator	Description
'+'	**Unary plus operator**; indicates positive value (numbers are positive without this, however)
'-'	**Unary minus operator**; negates an expression value
'++'	**Increment operator**; increments a value by 1
'--'	**Decrement operator**; decrements a value by 1
'!'	**Logical complement operator**; inverts the value of a boolean

3.2. Binary Arithmetic Operators

Operator	Description
'+'	**Addition** – Adds values on either side of the operator
'-'	**Subtraction** – Subtracts right hand operand from left hand operand
'*'	**Multiplication** – Multiplies values on either side of the operator
'/'	**Division** – Divides left hand operand by right hand operand
'%'	**Modulus** – Divides left hand operand by right hand operand and returns remainder

4. String Concatenation Operator

The '+' operator is overloaded in Java. An operator is said to be overloaded if it is used to perform more than one function.

4.1. Concatenating Two Strings

So far, you have seen its use as an arithmetic addition operator to add two numbers. It can also be used to **concatenate two strings**.

```
String str1 = "Hello";

String str2 = " World";



String str3 = str1 + str2;      // Assigns "Hello World" to str3
```

4.2. Concatenating Primitive Types to String

The string concatenation operator is also used to concatenate a primitive and a reference data type value to a string.

```
int num = 26;



String str1 = "Alphabets";



String str2 = num + str1;    // Assigns "26Alphabets" to str2
```

4.3. Concatenate *null*

If a reference variable contains the '*null*' reference, the concatenation operator uses a string "null".

```
String str1 = "I am ";
String str2 = null;



String str3 = str1 + str2;    // Assigns "I am null" to str3
```

5. Relational Operators

- All relational operators are binary operators.
- They take two operands.
- The result produced by a relational operator is always a Boolean value true or false.
- They are mostly used in Java control statements such as if statements, while statements etc.

Let's see below all available relational operators in java.

Operator	Description
'=='	**Equals to** – Checks if the values of two operands are equal or not, if yes then condition becomes tru
'!='	**Not equals to** – Checks if the values of two operands are equal or not, if values are not equal then c becomes true.
'>'	**Greater than** – Checks if the value of left operand is greater than the value of right operand, if yes t condition becomes true.
'<'	**Less than** – Checks if the value of left operand is less than the value of right operand, if yes then co becomes true.
'>='	**Greater than or equals to** – Checks if the value of left operand is greater than or equal to the value operand, if yes then condition becomes true.

'<='	**Less than or equals to** – Checks if the value of left operand is less than or equal to the value of righ yes then condition becomes true.

```
int result = 20;



if( result &gt; 10) {          //true
  //some operation
}

boolean isEqual = ( 10 == 20 );    //false
```

6. Boolean Logical Operators

- All Boolean logical operators can be used only with boolean operand(s).
- They are mostly used in control statements to compare two (or more) conditions.

Operator	Description
'!'	returns true if the operand is false, and false if the operand is true.
'&&'	returns true if both operands are true. If either operand is false, it returns false.
'&'	returns true if both operands are true. If either operand is false, it returns false.
'||'	returns true if either operand is true. If both operands are false, it returns false.
'|'	returns true if either operand is true. If both operands are false, it returns false.
'^'	it returns true if one of the operands is true, but not both. If both operands are the same, it returns

'&=;'	if both operands evaluate to true, &= returns true. Otherwise, it returns false.
'|='	if either operand evaluates to true, != returns true. Otherwise, it returns false.
'^='	if both operands evaluate to different values, that is, one of the operands is true but not both, ^= re Otherwise, it returns false.

```
int result = 20;



if( result > 10 && result < 30) {

   //some operation
}

if( result > 10 || result < 30) {
   //some operation
}
```

1. The **logical AND operator** (&) works the same way as the logical short-circuit AND operator (&&), except for one difference. The logical AND operator (&) evaluates its right-hand operand even if its left-hand operand evaluates to false.
2. The **logical OR operator** works the same way as the logical short-circuit OR operator, except for one difference. The logical OR operator evaluates its right-hand operand even if its left-hand operand evaluates to true.

7. Bitwise Operators

A bitwise operator **manipulates individual bits** of its operands. Java defines several bitwise operators, which can be applied to the integer types, *long*, *int*, *short*, *char*, and *byte*.

Operator	Description

'&'	**Binary AND Operator** copies a bit to the result if it exists in both operands.
'|'	**Binary OR Operator** copies a bit if it exists in either operand.
'^'	**Binary XOR Operator** copies the bit if it is set in one operand but not both.
'~'	**Binary Ones Complement Operator** is unary and has the effect of 'flipping' bits.
<<	**Binary Left Shift Operator**. The left operands value is moved left by the number of bits specified by operand.
>>	**Binary Right Shift Operator**. The left operands value is moved right by the number of bits specified operand.
>>>	**Shift right zero fill operator**. The left operands value is moved right by the number of bits specified operand and shifted values are filled up with zeros.

8. Ternary Operator

- Java has one conditional operator. It is called a ternary operator as it takes **three operands**.
- The two symbols of "?" and ":" make the ternary operator.
- If the boolean-expression evaluates to *true*, it evaluates the *true-expression*; otherwise, it evaluates false-expression.

boolean-expression ? true-expression : false-expression

```
int number1 = 40;

int number2 = 20;
int biggerNumber = (number1 > number2) ? number1 : number2;

//Compares both numbers and return which one is bigger
```

9. Operators Precedence Table

Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have higher precedence than addition and subtraction.

Precedence rules can be overridden by explicit parentheses.
When two operators share an operand the operator with the higher precedence goes first. For example, 1 + 2 * 3 is treated as 1 + (2 * 3) because the precedence of multiplication is higher than addition.

In the above expression, if you want to add values first then use explicit parentheses like this – (1 + 2) * 3.

Precedence	Operator	Type	Ass
15	() [] ·	Parentheses Array subscript Member selection	Left to
14	++ —	Unary post-increment Unary post-decrement	Right to
13	++ — + – !	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation	Right to

	~ (*type*)	Unary bitwise complement Unary type cast	
12	* / %	Multiplication Division Modulus	Left to
11	+ –	Addition Subtraction	Left to
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to
8	== !=	Relational is equal to Relational is not equal to	Left to
7	&	Bitwise AND	Left to
6	^	Bitwise exclusive OR	Left to
5	|	Bitwise inclusive OR	Left to
4	&&	Logical AND	Left to

3	||	Logical OR	Left to
2	? :	Ternary conditional	Right to
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to

That's all for the operators in java.

Q3.What is the use of Switch case in Java programming?

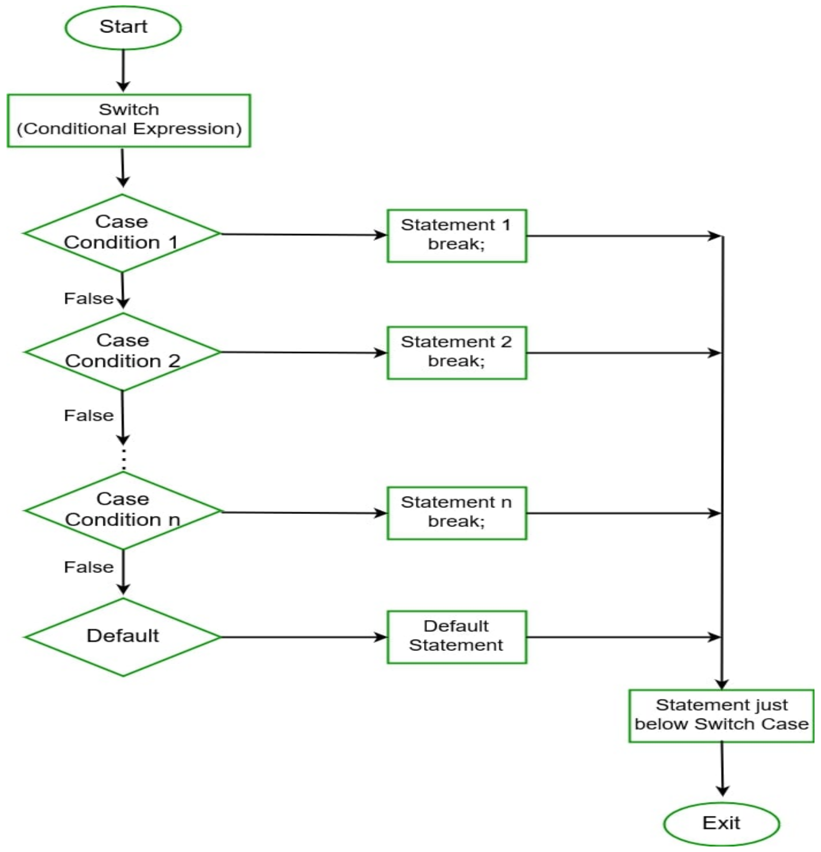
The **switch statement** is a multi-way branch statement. In simple words, the Java switch statement executes one statement from multiple conditions. It is like an if-else-if ladder statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression. Basically, the expression can be a byte, short, char, or int primitive data types. It basically tests the equality of variables against multiple values.
Note: *Java switch expression must be of byte, short, int, long(with its Wrapper type), enums and string. Beginning with JDK7, it also works with enumerated types (Enums in java), the String class, and Wrapper classes.*

Some Important Rules for Switch Statements

1. There can be any number of cases just imposing condition check but remember duplicate case/s values are not allowed.
2. The value for a case must be of the same data type as the variable in the switch.
3. The value for a case must be constant or literal. **Variables are not allowed.**
4. The break statement is used inside the switch to terminate a statement sequence.
5. The break statement is optional. If omitted, execution will continue on into the next case.
6. The default statement is optional and can appear anywhere inside the switch block. In case, if it is not at the end, then a break statement must be kept after the default statement to omit the execution of the next case statement.

Note: Until Java-6, switch case argument cannot be of String type but Java 7 onward we can use String type argument in Switch Case.

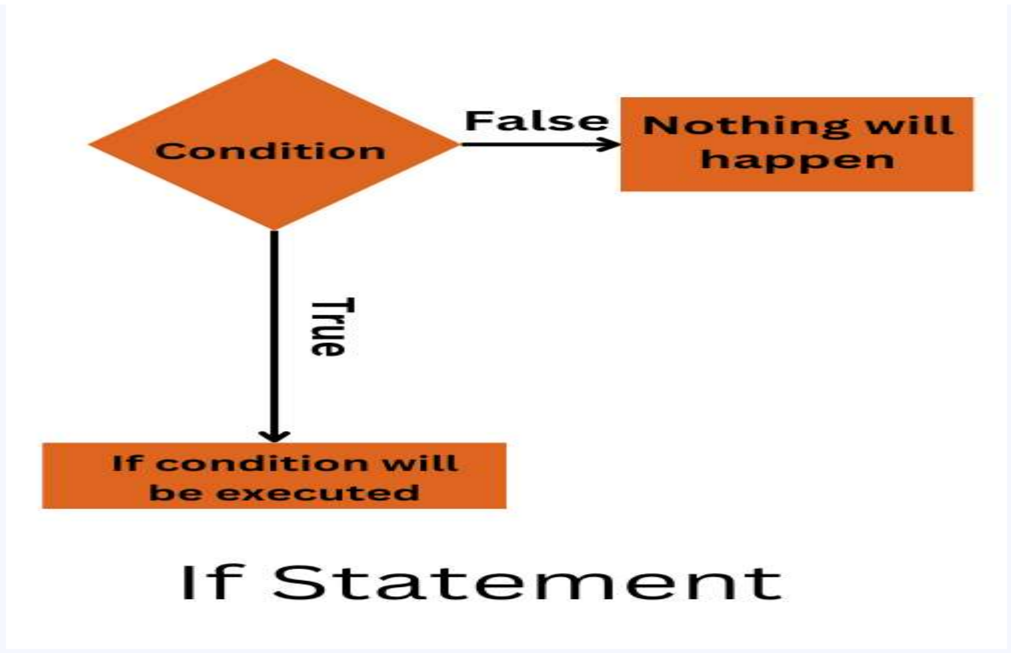
Flow Diagram of Switch-Case Statement



Q4.What are the conditional Statements and use of conditional statements in Java?

Conditional Statements in Java(If-Else Statement)
Introduction
In programming languages like the real world, we must make decisions based on some conditions to perform the tasks. Conditional statements in programming languages help us to make such decisions.
In this blog, we will discuss Conditional Statements in Java in detail. Let's start going!

What are Conditional Statements in Java?
The conditional statement is a branch of code that can be executed depending on another condition. In Java, these clauses are called decision or selection statements.
There are five types of Java conditional statements:-
1. Java If Statement
2. Java If-Else Statement
3. Java If-Else-If Ladder Statement
4. Java Nested If Statement
5. Java Switch Statement
Java If Statement



Suppose a condition is true **if a statement** is used to run the program. It is also known as a one-way selection statement. If a condition is used, an argument is passed, and if it is satisfied, the corresponding code is executed; otherwise, nothing happens.

Syntax

The syntax of the If statement is:-

```
if (expression) {
   // You can enter the code here
}
```

Working of Java If Statement

If conditional statements in java works on the following given statement:-

1. Statements in the block are executed if the expression evaluates to nonzero.
2. Control is transferred to the statement that follows the expression if it evaluates to zero (false).

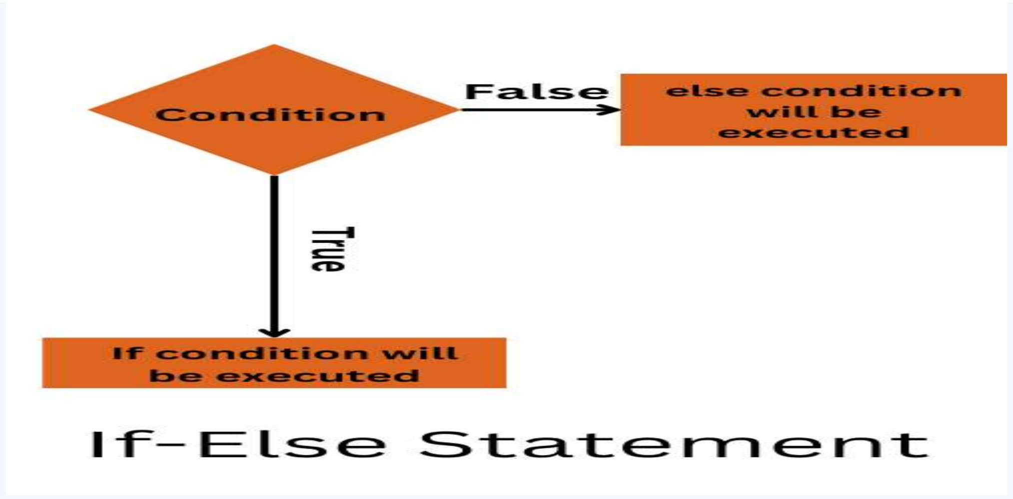
Example

The below given example helps in understanding the if statement in Java.

```
import java.util.*;
class sample {
   public static void main(String[] args) {
      int a = -3;
      if (a < 0)   {
         System.out.println(a + " is a Negative Number.");
      }
   }
}
```

Output:

Java If-Else Statement


Condition
False
else condition will be executed
True
If condition will be executed
If-Else Statement

An if-else statement is a control structure that determines which statements to choose based on a set of conditions.

Syntax

The syntax of the If-Else conditional statements in java is:-

```
if (condition) {
   // Statements that will be carried out if the condition is satisfied
} else {
   // Statements that will be carried out if the condition is not met
}
```

Working of Java If-Else Statement

The statements in the if block is carried out if the condition is met. Otherwise, the statements in the else block are carried out. Always follow an if clause with an else clause. If not, the compiler will produce a "misplaced else" error.

Example

The example below helps us understand the if-else conditional statements in Java.

```
import java.util.*;
class sample {
   public static void main(String[] args) {
      int a = 10;
      if (a % 2 == 0)    {
         System.out.println(a + " is an even number");
      }
      else    {
         System.out.println(a + " is an odd number");
      }
   }
}
```

Output:

Using Ternary Operator

The Java ternary operator offers a concise syntax for determining if a condition is true or false. It returns a value based on the outcome of the Boolean test.
One can replace their Java if-else statement using a ternary operator to create compact code. Expert coders enjoy the clarity and simplicity the Java ternary operator adds to their code.
The syntax of the Java Ternary operator is as follows:

```
(condition) ? (return if true) : (return if false);
```

Let's understand the concept with the help of a code example. We will see the above code using the ternary operator instead of the Java If-else statement.

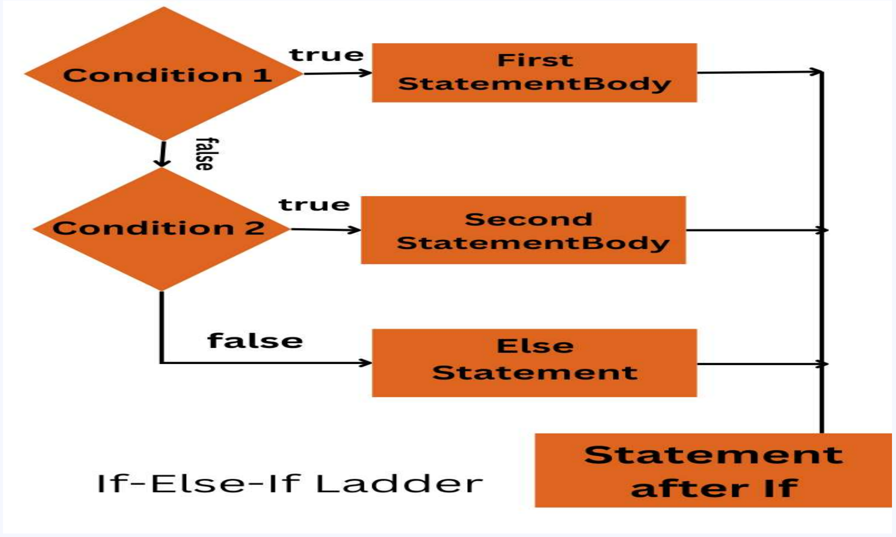
```
class example {
  public static void main( String args[] ) {
    int a = 10;
    String teropt =  (a % 2 == 0) ? a + " is an even number" : a + " is an odd number";
    System.out.println(teropt);
  }
}
```

Output

```
10 is an even number
```

Java If-Else-If Ladder Statement

An if-else-if ladder in Java can execute one code block while multiple other blocks are executed.



Syntax

The syntax of the If-Else-If conditional statements in java is:-

```
if (condition1) {
   // Executable code if condition1 is true,
} else if (condition2) {
   // Executable code if condition2 is true
}
```

```
...
else {
   // Executable code if all the conditions are false
}
```

Working of Java If-Else-If Ladder Statement

The conditional expressions are evaluated in the If-Else-If statement from top to bottom. The statement associated with a true condition is executed as soon as it is found, and the rest of the ladder is skipped. The final else statement is executed if all the conditions are false.

Example

The below given example helps in understanding the if-else-if ladder conditional statements in Java.

```
import java.util.*;
class sample {
   public static void main(String[] args) {
      int i = 3;
      if (i == 1)
          System.out.println("January");
      else if (i == 2)
          System.out.println("February");
      else if (i == 3)
          System.out.println("March");
      else
          System.out.println("April");
   }
}
```

Output:

Java Nested If Statement

If conditions inside another if conditions are called as Nested If conditional statements in java.

Syntax

The syntax of the nested if conditional statements in java are:-

```
if (condition1) {
   // Statement 1 will execute

   if (condition2)   {
     // Statement 2 will execute
   }
}
```

Working of Java Nested If Statement

The way nested if statements operate assumes that they must first become true and sufficient for the other states with the second condition for them to be used, even though other statements may choose to proceed with a false condition if the first condition is met.

Example
The example below helps us understand the Nested If conditional statements in Java.
```
import java.util.*;
class sample {
  public static void main(String[] args) {
    int i = 10;
    if (i == 10) {
      // First if statement
      if (i < 20)
        System.out.println("i is smaller than 20");
      // Nested - if statement
      if (i < 15)
        System.out.println("i is smaller than 15 too");
      else
        System.out.println("i is greater than 20");
    }
  }
}
```
Output:

Java Switch Statement
Unlike the if-else statement, the switch statement has more than one way to be executed. Additionally, it compares the expression's value to its cases by focusing its evaluation on a few primitive or class types.
Syntax
The syntax of the Switch statement is:-
```
switch (Expression) {
    case value 1:
        // Statement 1;
    case value 2:
        // Statement 2;
    case value 3:
        // Statement 3;

         ...
    case value n:
        // Statement n;
    Default:
        // default statement;
}
```
Working of Java Switch Statement
Java's switch case allows multiple conditions to be checked simultaneously, similar to an if-else ladder. A constant or a literal expression that can be evaluated is given to Switch.

Each test case's value is compared to the expression's value until a match is made. The associated code is executed if the specified default keyword does not find a match. Otherwise, the code designated for the test case that matches it is run.

Example

```
import java.util.*;
class Sample {
  public static void main(String[] args) {
    int mon = 2;
    switch (mon) {
    case 1:
      System.out.println("January");
      break;

    case 2:
      System.out.println("February");
      break;

    case 3:
      System.out.println("March");
      break;

    case 4:
      System.out.println("April");
      break;

    case 5:
      System.out.println("May");
      break;

    case 6:
      System.out.println("June");
      break;

    default:
      System.out.println("Not present in first half month of year");
    }
  }
}
```

Java If-Else vs Java Switch Statements

If-Else Statement	Switch Statement
Best for checking multiple conditions that can't be represented by a single expression	Best for checking a single expression against multiple values

If-Else Statement	Switch Statement
Evaluates a boolean expression to determine which branch to execute	Evaluates an expression to determine which case to execute
Can handle complex conditions with multiple boolean expressions	Can handle a limited set of values that can be matched
Can use all relational operators	Can only use the == operator to compare values

Q5.What is the syntax of if else statement?'

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
if(condition){
//code to be executed
}
```

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
```

```
    }
}
}
```

Test it Now

Output:

```
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

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

Example:

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Q6.How do you compare two strings in Java?

String is a sequence of characters. In Java, objects of String are immutable which means they are constant and cannot be changed once created.
Below are 5 ways to compare two Strings in Java:

1. **Using user-defined function :** Define a function to compare values with following conditions :
   1. if (string1 > string2) it returns a **positive value**.
   2. if both the strings are equal lexicographically i.e.(string1 == string2) it returns **0**.
   3. if (string1 < string2) it returns a **negative value**.

   The value is calculated as **(int)str1.charAt(i) – (int)str2.charAt(i)**
   Examples:
   Input 1: GeeksforGeeks
   Input 2: Practice
   Output: -9

   Input 1: Geeks
   Input 2: Geeks
   Output: 0

   Input 1: GeeksforGeeks
   Input 2: Geeks
   Output: 8
   Program:

```
// Java program to Compare two strings

// lexicographically

public class GFG {



   // This method compares two strings

   // lexicographically without using
// library functions

public static int stringCompare(String str1, String str2)

{



    int l1 = str1.length();

    int l2 = str2.length();

    int lmin = Math.min(l1, l2);



    for (int i = 0; i < lmin; i++) {

        int str1_ch = (int)str1.charAt(i);

        int str2_ch = (int)str2.charAt(i);



        if (str1_ch != str2_ch) {

            return str1_ch - str2_ch;

        }

    }



    // Edge case for strings like

    // String 1="Geeks" and String 2="Geeksforgeeks"
    if (l1 != l2) {

        return l1 - l2;

    }



    // If none of the above conditions is true,

    // it implies both the strings are equal

    else {

        return 0;

    }

}



// Driver function to test the above program

public static void main(String args[])

{

    String string1 = new String("Geeksforgeeks");

    String string2 = new String("Practice");

    String string3 = new String("Geeks");

    String string4 = new String("Geeks");
```

```
        // Comparing for String 1 < String 2

        System.out.println("Comparing " + string1 + " and " + string2

                       + " : " + stringCompare(string1, string2));



        // Comparing for String 3 = String 4

        System.out.println("Comparing " + string3 + " and " + string4

                       + " : " + stringCompare(string3, string4));



        // Comparing for String 1 > String 4

        System.out.println("Comparing " + string1 + " and " + string4

                       + " : " + stringCompare(string1, string4));

    }

}
```

Output:

```
Comparing Geeksforgeeks and Practice : -9

Comparing Geeks and Geeks : 0

Comparing Geeksforgeeks and Geeks : 8
```

2. **Using String.equals() :**In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are same then it returns true. If any character does not match, then it returns false.
   Syntax:
   str1.equals(str2);

   Here str1 and str2 both are the strings which are to be compared.

   Examples:
   Input 1: GeeksforGeeks

Input 2: Practice
Output: false

Input 1: Geeks
Input 2: Geeks
Output: true

Input 1: geeks
Input 2: Geeks
Output: false
Program:

```
// Java program to Compare two strings

// lexicographically

public class GFG {

    public static void main(String args[])

    {

        String string1 = new String("Geeksforgeeks");

        String string2 = new String("Practice");

        String string3 = new String("Geeks");

        String string4 = new String("Geeks");

        String string5 = new String("geeks");



        // Comparing for String 1 != String 2

        System.out.println("Comparing " + string1 + " and " + string2

                           + " : " + string1.equals(string2));
        // Comparing for String 3 = String 4

        System.out.println("Comparing " + string3 + " and " + string4

                      + " : " + string3.equals(string4));



        // Comparing for String 4 != String 5

        System.out.println("Comparing " + string4 + " and " + string5

                      + " : " + string4.equals(string5));



        // Comparing for String 1 != String 4

        System.out.println("Comparing " + string1 + " and " + string4

                      + " : " + string1.equals(string4));

    }

}
```

Output:

```
Comparing Geeksforgeeks and Practice : false

Comparing Geeks and Geeks : true

Comparing Geeks and geeks : false

Comparing Geeksforgeeks and Geeks : false
```

3. **Using String.equalsIgnoreCase() :** The String.equalsIgnoreCase() method compares two strings irrespective of the case (lower or upper) of the string. This method returns true if the argument is not null and the contents of both the Strings are same ignoring case, else false.
   Syntax:

str2.equalsIgnoreCase(str1);

Here str1 and str2 both are the strings which are to be compared.

Examples:
Input 1: GeeksforGeeks
Input 2: Practice
Output: false

Input 1: Geeks
Input 2: Geeks
Output: true

Input 1: geeks
Input 2: Geeks
Output: true
Program:

```
// Java program to Compare two strings

// lexicographically

public class GFG {

    public static void main(String args[])

    {

        String string1 = new String("Geeksforgeeks");

        String string2 = new String("Practice");

        String string3 = new String("Geeks");

        String string4 = new String("Geeks");

        String string5 = new String("geeks");
        // Comparing for String 1 != String 2

        System.out.println("Comparing " + string1 + " and " + string2

                          + " : " + string1.equalsIgnoreCase(string2));



        // Comparing for String 3 = String 4

        System.out.println("Comparing " + string3 + " and " + string4

                          + " : " + string3.equalsIgnoreCase(string4));



        // Comparing for String 4 = String 5

        System.out.println("Comparing " + string4 + " and " + string5

                          + " : " + string4.equalsIgnoreCase(string5));



        // Comparing for String 1 != String 4

        System.out.println("Comparing " + string1 + " and " + string4

                          + " : " + string1.equalsIgnoreCase(string4));

    }

}
```

Output:

```
Comparing Geeksforgeeks and Practice : false

Comparing Geeks and Geeks : true

Comparing Geeks and geeks : true
```

Comparing Geeksforgeeks and Geeks : false

4. **Using Objects.equals() :** Object.equals(Object a, Object b) method returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.
   Syntax:
   public static boolean equals(Object a, Object b)

   Here a and b both are the string objects which are to be compared.

   Examples:
   Input 1: GeeksforGeeks
   Input 2: Practice
   Output: false

   Input 1: Geeks
   Input 2: Geeks
   Output: true

   Input 1: null
   Input 2: null
   Output: true
   Program:

```
// Java program to Compare two strings

// lexicographically



import java.util.*;



public class GFG {

    public static void main(String args[])

    {
```

```
String string1 = new String("Geeksforgeeks");

String string2 = new String("Geeks");

String string3 = new String("Geeks");

String string4 = null;

String string5 = null;



// Comparing for String 1 != String 2

System.out.println("Comparing " + string1 + " and " + string2

                 + " : " + Objects.equals(string1, string2));



// Comparing for String 2 = String 3

System.out.println("Comparing " + string2 + " and " + string3

                 + " : " + Objects.equals(string2, string3));



// Comparing for String 1 != String 4

System.out.println("Comparing " + string1 + " and " + string4

                 + " : " + Objects.equals(string1, string4));



// Comparing for String 4 = String 5
```

```
        System.out.println("Comparing " + string4 + " and " + string5

                            + " : " + Objects.equals(string4, string5));

    }

}
```

Output:

```
Comparing Geeksforgeeks and Geeks : false

Comparing Geeks and Geeks : true

Comparing Geeksforgeeks and null : false

Comparing null and null : true
```

5. Using String.compareTo() **:**

   Syntax:
   int str1.compareTo(String str2)

   Working:
   It compares and returns the following values as follows:
   1. if (string1 > string2) it returns a **positive value**.
   2. if both the strings are equal lexicographically i.e.(string1 == string2) it returns **0**.
   3. if (string1 < string2) it returns a **negative value**.

   Examples:
   Input 1: GeeksforGeeks
   Input 2: Practice
   Output: -9

   Input 1: Geeks
   Input 2: Geeks
   Output: 0

   Input 1: GeeksforGeeks
   Input 2: Geeks
   Output: 8
   Program:

```
// Java program to Compare two strings
// lexicographically


import java.util.*;


public class GFG {

    public static void main(String args[])

    {

        String string1 = new String("Geeksforgeeks");

        String string2 = new String("Practice");

        String string3 = new String("Geeks");

        String string4 = new String("Geeks");


        // Comparing for String 1 < String 2

        System.out.println("Comparing " + string1 + " and " + string2

                           + " : " + string1.compareTo(string2));


        // Comparing for String 3 = String 4

        System.out.println("Comparing " + string3 + " and " + string4

                           + " : " + string3.compareTo(string4));
        // Comparing for String 1 > String 4

        System.out.println("Comparing " + string1 + " and " + string4

                           + " : " + string1.compareTo(string4));

    }

}
```

Q7.What is Mutable String in Java Explain with an example

Introduction to Mutable String in Java

With Mutable string, we can change the contents of an existing object, which does not create a new object. Therefore mutable strings are those strings whose content can be changed without creating a new object. StringBuffer and StringBuilder are mutable versions of String in java, whereas the java String class is immutable. Immutable objects are those objects whose contents cannot be modified once created. Whenever an immutable object's content is changed, there will be a creation of a new object.

How to Use Mutable String in Java?

As already covered, the mutable string in java can be created using StringBuffer and StringBuilder classes. The main difference between the two is that StringBuffer is a thread-safe implementation, whereas StringBuilder is not. Whenever high performance and high security is required, one should prefer mutable versions of the String class. Because of the String pool, there are security issues with the String class; therefore, StringBuffer and StringBuilder are used whenever data security is necessary. StringBuffer is better in terms of performance than StringBuffer as StringBuffer is thread-safe, but whenever thread safety is necessary, one should go for StringBuffer.

Constructors of StringBuilder and StringBuffer Classes
The following are the main constructors of string builder and string buffer classes.

1. StringBuffer Constructors
The following are String buffer constructors:

- **StringBuffer():** This creates an empty StringBuffer with a default capacity of 16 characters.
- **StringBuffer(int capacity):** This creates an empty StringBuffer with a specified capacity.
- **StringBuffer(CharSequence charseq):** This creates StringBuffer containing the same characters as in the specified character sequence.
- **StringBuffer(String str):** Creates a StringBuffer corresponding to specified String.

Here is the declaration of StringBuffer Class:

```
public final class StringBuffer extends Object implements Serializable,CharacterSequence
```

Q8.Write a program to sort a String Alphabetically ?

Algorithm

- Take a String input from user and store it in the variable called **“s” (in this case)**

- After that convert **String to character array** using **tocharArray()** method
- Use **Arrays class sort method** to **sort()** the character array
- At last print sorted array **System.out.println(c)**

Code in Java

```
import java.util.Arrays;
import java.util.Scanner;
public class SortAStringAlphabetically {

public static void main(String[] args){
   Scanner sc = new Scanner(System.in);
   System.out.print("Enter String :");
   String s = sc.nextLine();
   char[] c = s.toCharArray();
   Arrays.sort(c);
   System.out.println(c);
  }
}
```

Output

```
Enter String :Avanish
Aahinsv
```

Q9.Write a program to check if the letter 'e' is present in the word

'Umbrella'.

```
class Check_Letter
{
        public static void main(String[] args)
        {
                String str = "Umbrella";
                boolean pre = false;
                for(int i = 0;i<str.length();i++)
                {
                        if(str.charAt(i) == 'e')
                        {
                                pre=true;
```

```
                                        break;
                                }
                        }
                        System.out.println(pre);
                }
}
```

Output

```
True
```

Q10.Where exactly is the string constant pool located in the

memory?

String pool is a storage space in the Java heap memory where string literals are stored. It is also known as String Constant Pool or String Intern Pool. It is privately maintained by the Java String class. By default, the String pool is empty. A pool of strings decreases the number of String objects created in the JVM, thereby reducing memory load and improving performance.

Scope

The article aims to:

- Help understand the concept and need of String Constant Pool in Java
- Explain changes in the String Pool as a result of creating new strings using the new keyword and String.intern() method
- Discuss the advantages and disadvantages of String Pool in Java

Introduction

A string in Java is a **set of characters** which is always enclosed in double quotes. A string can also be described alternatively as an array of characters. But in addition, strings in Java are treated as **objects**.

Strings are **immutable** in nature. Immutable means they have a **constant value**, and even if they are altered, instead of reflecting the alterations in the original string, a new object is created. This immutability is achieved through **String Pool**. Let us look at the concept of String Pool.

What is String Pool in Java?

String Pool in Java is a special storage space in Java *Heap memory where string literals are stored.* It is also known by the names - **String Constant Pool** or **String Intern Pool**. Whenever a string literal is created, the JVM first checks the String Constant Pool before creating a new String object corresponding to it.

Let us first discuss the memory allocation methods used in Java - **Stack Memory Allocation** and **Heap Memory Allocation**.

- In stack memory, only the primitive data types like- int, char, byte, short, boolean, long, float and double are stored.
- Whereas, in the heap memory, non-primitive data types like strings are stored. A reference to this location is held by the stack memory.

Explanation:

- Since int and double are primitive data types, they are stored in the stack memory itself.
- However, idName stores a String that is non-primitive in nature. Hence, the String object is created in the heap memory, and its reference is stored by idName in the stack memory.

Memory Allocation in the String Pool

- The String Pool is empty by default, and it is maintained privately by the String class.
- When we create a string literal, the JVM first checks that literal in the String Constant Pool. If the literal is already present in the pool, its reference is stored in the variable.
- However, if the string literal is not found, the JVM creates a new string object in the String Constant Pool and returns its reference.

Need of String Constant Pool:

- When we create a String object, it uses some amount of space in the heap memory.
- Let's say we are creating n number of String objects with the same value, and distinct memory is allocated to each of these string objects (though they all contain the same string).
- This is an inefficient usage of heap memory. In order to escalate the performance of our code and reduce memory usage, JVM optimizes the way in which strings are stored with the help of a string constant pool.

Ways to Create Strings

There are three popular ways of creating strings in Java:

- String literal
- Using new keyword
- Using String.intern() method