Q. What is an Operator? Discuss the available operators in c# Language.

Ans:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators:

These are used to perform arithmetic/mathematical operations on operands. The **Binary Operators** falling in this category are :

- **Addition:** The **'+'** operator adds two operands. For example, **x+y**.
- **Subtraction:** The **'-'** operator subtracts two operands. For example, **x-y**.
- **Multiplication:** The **'*'** operator multiplies two operands. For example, **x*y**.
- **Division:** The **'/'** operator divides the first operand by the second. For example, **x/y**.
- **Modulus:** The **'%'** operator returns the remainder when first operand is divided by the second. For example, **x%y**.

Relational Operators:

Relational operators are used for comparison of two values. Let's see them one by one:

- **'=='(Equal To)** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false. For example, **5==5** will return true.
- **'!='(Not Equal To)** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false. It is the exact boolean complement of the **'=='** operator. For example, **5!=5** will return false.
- **'>'(Greater Than)** operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false. For example, **6>5** will return true.
- **'<'(Less Than)** operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false. For example, **6<5** will return false.
- **'>='(Greater Than Equal To)** operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, **5>=5** will return true.
- **'<='(Less Than Equal To)** operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, **5<=5** will also return true.

Logical Operators:

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. They are described below:

- **Logical AND:** The **'&&'** operator returns true when both the conditions in consideration are satisfied. Otherwise it returns false. For example, **a && b** returns true when both a and b are true (i.e. non-zero).
- **Logical OR:** The **'||'** operator returns true when one (or both) of the conditions in consideration is satisfied. Otherwise it returns false. For example, **a || b** returns true if one of a or b is true (i.e. non-zero). Of course, it returns true when both a and b are true.
- **Logical NOT:** The **'!'** operator returns true the condition in consideration is not satisfied. Otherwise it returns false. For example, **!a** returns true if a is false, i.e. when a=0.

Bitwise Operators:

In C#, there are 6 bitwise operators which work at bit level or used to perform bit by bit operations. Following are the bitwise operators :

- **& (bitwise AND)** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- **| (bitwise OR)** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
- **^ (bitwise XOR)** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- **<< (left shift)** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- **>> (right shift)** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

Assignment operators:

Assignment operators are used to assigning a value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error. Different types of assignment operators are shown below:

- **"="(Simple Assignment)** : This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
  Example :

```
a = 10;
b = 20;
ch = 'y';
```

- **"+="(Add Assignment)** : This operator is combination of '+' and '=' operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example :

```
(a += b) can be written as (a = a + b)
```

  If initially value stored in a is 5. Then (a += 6) = 11.

- **“-=”(Subtract Assignment)** : This operator is combination of ‘-‘ and ‘=’ operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a -= b) can be written as (a = a - b)`

  If initially value stored in a is 8. Then (a -= 6) = 2.

- **“*=”(Multiply Assignment)** : This operator is combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a *= b) can be written as (a = a * b)`

  If initially value stored in a is 5. Then (a *= 6) = 30.

- **“/=”(Division Assignment)** : This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a /= b) can be written as (a = a / b)`

  If initially value stored in a is 6. Then (a /= 2) = 3.

- **“%=”(Modulus Assignment)** : This operator is combination of ‘%’ and ‘=’ operators. This operator first modulo the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a %= b) can be written as (a = a % b)`

  If initially value stored in a is 6. Then (a %= 2) = 0.

- **“<<="(Left Shift Assignment)** : This operator is combination of ‘<<' and '=' operators. This operator first Left shift the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a <<= 2) can be written as (a = a << 2)`

  If initially value stored in a is 6. Then (a <<= 2) = 24.

- **“>>=”(Right Shift Assignment)** : This operator is combination of ‘>>’ and ‘=’ operators. This operator first Right shift the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a >>= 2) can be written as (a = a >> 2)`

  If initially value stored in a is 6. Then (a >>= 2) = 1.

- **“&=”(Bitwise AND Assignment)** : This operator is combination of ‘&’ and ‘=’ operators. This operator first “Bitwise AND” the current value of the variable on the left by the value on the right and then assigns the result to the variable on the left.
  Example :
- `(a &= 2) can be written as (a = a & 2)`

  If initially value stored in a is 6. Then (a &= 2) = 2.

- **“^=”(Bitwise Exclusive OR)** : This operator is combination of ‘^’ and ‘=’ operators. This operator first “Bitwise Exclusive OR” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :

```
(a ^= 2) can be written as (a = a ^ 2)
```

If initially value stored in a is 6. Then (a ^= 2) = 4.

- **"|="(Bitwise Inclusive OR)** : This operator is combination of '|' and '=' operators. This operator first "Bitwise Inclusive OR" the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example :
-
```
(a |= 2) can be written as (a = a | 2)
```

If initially value stored in a is 6. Then (a |= 2) = 6.

……

Q.2 Define conditional Statement in c # Language.
Ans:

It is ternary operator which is a shorthand version of if-else statement. It has three operands and hence the name ternary. It will return one of two values depending on the value of a Boolean expression.

Syntax :

condition ? first_expression : second_expression;

Explanation :

condition: It must be evaluate to true or false.

If the condition is true

first_expression is evaluated and becomes the result.

If the condition is false,

second_expression is evaluated and becomes the result.

Q.3. What we mean by Operator Precedence? Discuss.

Ans :

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so the first evaluation takes place for 3*2 and then 7 is added into it.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Q. 4. What is decision making statements? Discuss .

Ans :

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

C# provides following types of decision making statements:

if statement:

An **if** statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in C# is −

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true
*/
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement is executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement(after the closing curly brace) is executed.

If else statement:

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

The syntax of an **if...else** statement in C# is −

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true
*/
} else {
   /* statement(s) will execute if the boolean expression is
false */
}
```

If the boolean expression evaluates to **true**, then the **if block** of code is executed, otherwise **else block** of code is executed.

nested if statement:

It is always legal in C# to **nest** if-else statements, which means we can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows −

```
if( boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2) {
      /* Executes when the boolean expression 2 is true */
   }
```

```
}
```

We can nest **else if...else** in the similar way as we have nested *if* statement.

switch statement:

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C# is as follows −

```
switch(expression)
 {
   case constant-expression1  :
      statement(s);
      break;
   case constant-expression2  :
   case constant-expression3  :
      statement(s);
      break;

   /* we can have any number of case statements */
   default : /* Optional */
   statement(s);
}
```

The following rules apply to a **switch** statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, then it will raise a compile time error.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true.

Nested switch statement:

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows −

```
switch(ch1)
{
   case 'A':
   Console.WriteLine("This A is part of outer switch" );

   switch(ch2)
 {
      case 'A':
         Console.WriteLine("This A is part of inner switch" );
         break;
      case 'B': /* inner B case code */
   }
   break;
   case 'B': /* outer B case code */
}
```

.....

Q. 5. What we mean Loops? Discuss the loops available in C # Language.

Ans:

There may be a situation, when we need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages –

C# provides following types of loop to handle looping requirements.

While loop :

It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. or

A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.

Syntax:

The syntax of a **while** loop in C# is −

```
while(condition)
 {
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

For Loop:

It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. Or

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C# is −

```
for ( init; condition; increment )
{
   statement(s);
}
```

Here is the flow of control in a for loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again testing for a condition). After the condition becomes false, the for loop terminates.
-

Do While Loop:

It is similar to a while statement, except that it tests the condition at the end of the loop body

Unlike **for** and **while** loops, which test the loop condition at the start of the loop, the **do...while** loop checks its condition at the end of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a **do...while** loop in C# is −

```
do
{
   statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

For Each Loop:

The **foreach loop** is used to iterate over the elements of the collection. The collection may be an array or a list. It executes for each element present in the array.

- It is necessary to enclose the statements of foreach loop in curly braces {}.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.

Syntax:

```
foreach(data_type var_name in collection_variable)

{

   // statements to be executed

}
```

……

Q. 6. Discuss the Break, Continue and Go To Statement in C # Language.

Ans :

In C#, Jump statements are used to transfer control from one point to another point in the program due to some specified code while executing the program.

Break statement :

The break statement is used to terminate the loop or statement in which it present. After that, the control will pass to the statements that present after the break statement, if available. If the break statement present in the nested loop, then it terminates only those loops which contains break statement.

```
using System;

class Geeks
{

    static public void Main()
    {

        for (int i = 1; i < 4; i++)
        {
            if (i == 3)
                break;

            Console.WriteLine("GeeksforGeeks");
        }
    }
}
```

continue statement;

This statement is used to skip over the execution part of the loop on a certain condition. After that, it transfers the control to the beginning of the loop. Basically, it skips its following statements and continues with the next iteration of the loop.

```
using System;

class Geeks
{
    public static void Main()
    {
        for (int i = 1; i <= 10; i++)
       {
              if (i == 4)
                continue;

            Console.WriteLine(i);
        }
    }
}
```

goto statement:

This statement is used to transfer control to the labeled statement in the program. The label is the valid identifier and placed just before the statement from where the control is transferred.

```
using System;

class Geeks
{
      static public void Main()
    {
        int number = 20;
        switch (number)
{
          case 5:
            Console.WriteLine("case 5");
            break;
        case 10:
            Console.WriteLine("case 10");
            break;
        case 20:
            Console.WriteLine("case 20");
              goto case 5;
        default:
            Console.WriteLine("No match found");
            break;
        }
    }
}
```