Session: **3**

Statements and Operators

.net

◆ Define and describe statements and expressions

◆ Explain the types of operators

◆ Explain the process of performing data conversions in C#

- A C# program is a set of tasks that perform to achieve the overall functionality of the program.

- To perform the tasks, programmers provide instructions. These instructions are called statements.

- A C# statement can contain expressions that evaluates to a value.

- Statements are referred to as logical grouping of variables, operators, and C# keywords that perform a specific task.

- For example, the line which initializes a variable by assigning it a value is a statement.

- In C#, a statement ends with a semicolon.

- A C# program contains multiple statements grouped in blocks. A block is a code segment enclosed in curly braces.

- For example, the set of statements included in the `Main()` method of a C# code is a block.

- The following figure displays an example of a block of statements:

```
class Circle
{
    static void Main(string[] args)
    {
        const float _pi = 3.14F;
        float radius = 5;
        float area = _pi * radius * radius;
        Console.WriteLine("Area of the circle is " + area);
    }
}
```

- Statements are used to specify the input, the process, and the output tasks of a program. Statements can consist of:

    - Data types

    - Variables

    - Operators

    - Constants

    - Literals

    - Keywords

    - Escape sequence characters

- Statements help you build a logical flow in the program. With the help of statements, you can:

    - Initialize variables and objects

    - Take the input

    - Call a method of a class

    - Display the output

◆ The following code shows an example of a statement in C#:

Snippet	`double area = 3.1452 * radius * radius;`

◆ This line of code is an example of a C# statement that calculates the area of the circle and stores the value in the variable `area`.

◆ The following code shows an example of a block of statements in C#.

| Snippet | ```
{
    int side = 10;
    int height = 5;
    double area = 0.5 * side * height;
    Console.WriteLine("Area: " , area);
}
``` |
|---------|-----|

◆ In the code:

   ◈ A block of code is enclosed within curly braces.

   ◈ The first statement from the top will be executed first followed by the next statement and so on.

- The following code shows an example of nested blocks in C#:

Snippet

```
{
    int side = 5;
    int height = 10;
    double area;
    {
        area = 0.5 * side * height;
    }
    Console.WriteLine(area);
}
```

- In the code:
  - Another block of code is nested within a block of statements.
  - The first three statements from the top will be executed in sequence.
  - Then, the line of code within the inside braces will be executed to calculate the area.
  - The execution is terminated at the last statement in the block of the code displaying the area.

◆ Similar to statements in C and C++, the C# statements are classified into seven categories:

- ◈ Selection Statements

- ◈ Iteration Statements

- ◈ Jump Statements

- ◈ Exception Handling Statements

- ◈ Checked and Unchecked Statements

- ◈ Fixed Statement

- ◈ Lock Statement

- Following are the features of the checked and unchecked statements:

The checked statement checks for an arithmetic overflow in arithmetic expressions and the unchecked statement does not check for an arithmetic overflow.

An arithmetic overflow occurs if the result of an expression or a block of code is greater than the range of the target variable's data type causing the program to throw an exception that is caught by the `OverflowException` class.

Exceptions are runtime errors that disrupt the normal flow of the program.

The `System.Exception` class is used to derive several exception classes that handle the different types of exceptions.

The checked statement is associated with the `checked` keyword. When an arithmetic overflow occurs, the checked statement halts the execution of the program.

- A checked statement creates a checked context for a block of statements and has the following form:

```
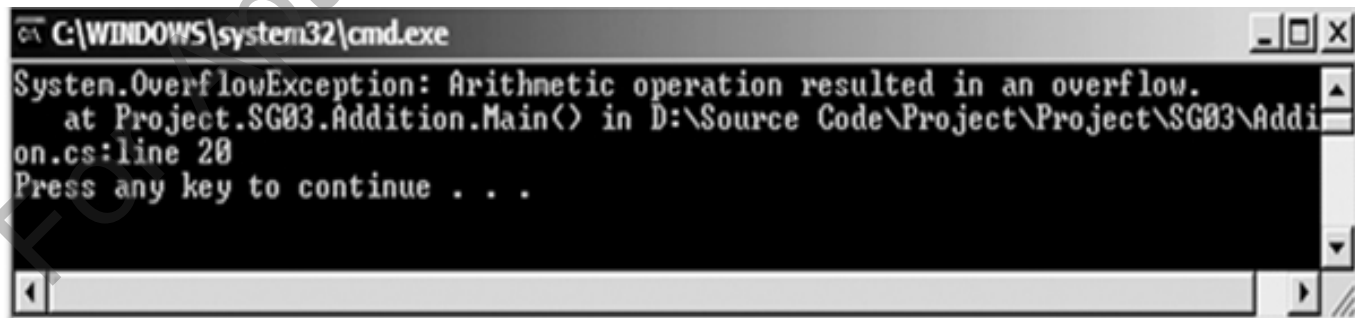checked-statement:
checked block
```

◆ The following code creates a class **Addition** with the checked statement that throws an overflow exception.

Snippet

```
using System;
class Addition
{
    public static void Main()
    {
        byte numOne = 255;
        byte numTwo = 1;
        byte result = 0;
        try
        {
            //This code throws an overflow
            // exception checked
            {
             result = (byte)(numOne + numTwo);
            }
            Console.WriteLine("Result: " +
            result);
        }
        catch (OverflowException ex)
        {
            Console.WriteLine(ex);
        }
    }
}
```

When the statement in the checked block is executed, it gives an error. This is because the result of the addition of the two numbers results in 256. This value is too large to be stored in the byte variable causing an arithmetic overflow to occur.

Output

```
C:\WINDOWS\system32\cmd.exe                              _ □ X
System.OverflowException: Arithmetic operation resulted in an overflow.
    at Project.SG03.Addition.Main() in D:\Source Code\Project\Project\SG03\Addi
on.cs:line 20
Press any key to continue . . .
```

- The unchecked statement is associated with the unchecked keyword.

- The unchecked statement ignores the arithmetic overflow and assigns junk data to the target variable.

- An unchecked statement creates an unchecked context for a block of statements and has the following form:

```
unchecked-statement:
unchecked block
```

◆ The following code creates a class **Addition** that uses the unchecked statement:

Snippet

```
using System;
class Addition
{
    public static void Main()
    {
        byte numOne = 255;
        byte numTwo = 1;
        byte result = 0;
        try
        {
          unchecked
          {
              result = (byte)(numOne + numTwo);
          }
          Console.WriteLine("Result: " + result);
        }
        catch (OverflowException ex)
        {
           Console.WriteLine(ex);
        }
    }

}
```

◆ In the code:

  ◈ When the statement within the unchecked block is executed, the overflow that is generated is ignored by the unchecked statement and it returns an unexpected value.

  ◈ The checked and unchecked statements are similar to the checked and unchecked operators.

  ◈ The only difference is that the statements operate on blocks instead of expressions.

- Expressions are used to manipulate data. Like in mathematics, expressions in programming languages, including C#, are constructed from the operands and operators.

- An expression statement in C# ends with a semicolon (;).

- Expressions are used to:
  - Produce values.
  - Produce a result from an evaluation.
  - Form part of another expression or a statement.

- The following code demonstrates an example for expressions:

Snippet

```
simpleInterest = principal * time * rate / 100;
eval = 25 + 6 – 78 * 5;

num++;
```

- In the first two lines of code, the results of the statements are stored in the variables `SimpleInterest` and `eval`. The last statement increments the value of the variable `num`.

- Some fundamental differences between statements and expressions are listed in the table.

Statements	Expressions
Do not necessarily return values. For example, consider the following statement: `int oddNum = 5;` The statement only stores the value 5 in the `oddNum` variable.	Always evaluates to a value. For example, consider the following expression: `100*(25*10)` The expression evaluates to the value 2500.
Statements are executed by the compiler.	Expressions are part of statements and are evaluated by the compiler.

◆ Following are the features of operators in C#:

Expressions in C# comprise one or more operators that performs some operations on variables.

An operation is an action performed on single or multiple values stored in variables in order to modify them or to generate a new value with the help of minimum one symbol and a value.

The symbol is called an operator and it determines the type of action to be performed on the value.

An operand might be a complex expression. For example, `(X * Y) + (X - Y)` is a complex expression, where the + operator is used to join two operands.

The value on which the operation is to be performed is called an operand.

- Operators are used to simplify expressions.

- In C#, there is a predefined set of operators used to perform various types of operations.

- These are classified into six categories based on the action they perform on values:

  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Conditional Operators
  - Increment and Decrement Operators
  - Assignment Operators

()

+, -, ++, --, ~, !

*, /, %, +, -, >>, <<

>, >=, <, <=, ==, !=

&, ^, |, &&, ||,

?:, =, *=, /=, +=, -=

- Arithmetic operators are binary operators because they work with two operands, with the operator being placed in between the operands.

- These operators allow you to perform computations on numeric or string data.

- The following table lists the arithmetic operators along with their descriptions and an example of each type:

Operators	Description	Examples
+ (Addition)	Performs addition. If the two operands are strings, then it functions as a string concatenation operator and adds one string to the end of the other.	40 + 20
- (Subtraction)	Performs subtraction. If a greater value is subtracted from a lower value, the resultant output is a negative value.	100 - 47
* (Multiplication)	Performs multiplication.	67 * 46
/ (Division)	Performs division. The operator divides the first operand by the second operand and gives the quotient as the output.	12000 / 10
% (Modulo)	Performs modulo operation. The operator divides the two operands and gives the remainder of the division operation as the output.	100 % 33

- The following code demonstrates how to use the arithmetic operators:

Snippet

```
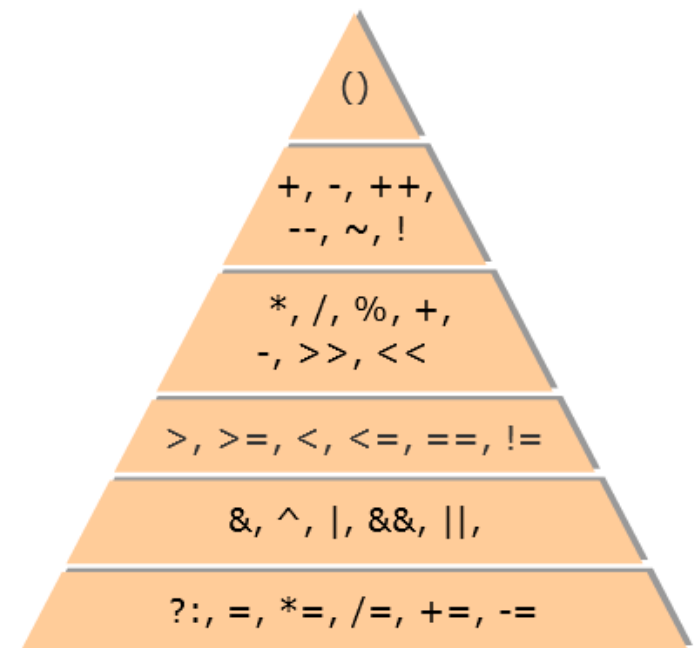int valueOne = 10;
int valueTwo = 2;
int add = valueOne + valueTwo;
int sub = valueOne - valueTwo;
int mult = ValueOne * valueTwo;
int div = valueOne / valueTwo;
int modu = valueOne % valueTwo;
Console.WriteLine("Addition " + add );
Console.WriteLine("Subtraction " + sub);
Console.WriteLine("Multiplication " + mult);
Console.WriteLine("Division " + div);

Console.WriteLine("Remainder " + modu);
```

Output

```
Addition 12
Subtraction 8
Multiplication 20
Division 5
Remainder 0
```

- Relational operators make a comparison between two operands and return a boolean value, true, or false.

- The following table lists the relational operators along with their descriptions and an example of each type.

Relational Operators	Description	Examples
==	Checks whether the two operands are identical.	85 == 95
!=	Checks for inequality between two operands.	35 != 40
>	Checks whether the first value is greater than the second value.	50 > 30
<	Checks whether the first value is lesser than the second value.	20 < 30
>=	Checks whether the first value is greater than or equal to the second value.	100 >= 30
<=	Checks whether the first value is lesser than or equal to the second value.	75 <= 80

- The following code demonstrates how to use the relational operators.

Snippet

```
int leftVal = 50;
int rightVal = 100;
Console.WriteLine("Equal: " + (leftVal == rightVal));
Console.WriteLine("Not Equal: " + (leftVal != rightVal));
Console.WriteLine("Greater: " + (leftVal > rightVal));
Console.WriteLine("Lesser: " + (leftVal < rightVal));
Console.WriteLine("Greater or Equal: " + (leftVal >= rightVal));

Console.WriteLine("Lesser or Equal: " + (leftVal <= rightVal));
```

Output

```
Equal: False
Not Equal: True
Greater: False
Lesser: True
Greater or Equal: False
Lesser or Equal: True
```

◆ Logical operators are binary operators that perform logical operations on two operands and return a boolean value.

◆ C# supports two types of logical operators:

Boolean Logical Operators

Bitwise Logical Operators

Boolean Logical Operators:

- Boolean logical operators perform boolean logical operations on both the operands. They return a boolean value based on the logical operator used.
- The following table lists the boolean logical operators along with their descriptions and an example of each type:

Logical Operators	Description	Examples
& (Boolean AND)	Returns true if both the expressions are evaluated to true.	`(percent >= 75) & (percent <= 100)`
| (Boolean Inclusive OR)	Returns true if at least one of the expressions is evaluated to true.	`(choice == 'Y') | (choice == 'y')`
^ (Boolean Exclusive OR)	Returns true if only one of the expressions is evaluated to true. If both the expressions evaluate to true, the operator returns false.	`(choice == 'Q') ^ (choice == 'q')`

◆ The following code explains the use of the boolean inclusive OR operator:

| Snippet | ```csharp
if ((quantity > 2000) | (price < 10.5))
{
    Console.WriteLine ("You can buy more goods at a lower price");
}
``` |

◆ In the code:

◈ The boolean inclusive OR operator checks both the expressions.

◈ If either one of them evaluates to true, the complete expression returns true and the statement within the block is executed.

◆ The following code explains the use of the boolean AND operator:

| Snippet | ```csharp
if ((quantity == 2000) & (price == 10.5))
{
    Console.WriteLine ("The goods are correctly priced");
}
``` |

◆ In the code:

◈ The boolean AND operator checks both the expressions.

◈ If both the expressions evaluate to true, the complete expression returns true and the statement within the block is executed.

◆ The following code explains the use of the boolean exclusive OR operator:

Snippet

```
if ((quantity == 2000) ^ (price == 10.5))
{
    Console.WriteLine ("You have to compromise between
    quantity and price");
}
```

◆ In the code:

◈ The boolean exclusive OR operator checks both the expressions.

◈ If only one of them evaluates to true, the complete expression returns true and the statement within the block is executed.

◈ If both of them are true, the expression returns false.

◆ **Bitwise Logical Operators:**

◈ The bitwise logical operators perform logical operations on the corresponding individual bits of two operands.

◈ The following table lists the bitwise logical operators along with their descriptions and an example of each type:

Logical Operators	Description	Examples
& (Bitwise AND)	Compares two bits and returns 1 if both bits are 1, else returns 0.	00111000 & 00011100
| (Bitwise Inclusive OR)	Compares two bits and returns 1 if either of the bits is 1.	00010101 | 00011110
^ (Bitwise Exclusive OR)	Compares two bits and returns 1 if only one of the bits is 1.	00001011 ^ 00011110

◈ The following code explains the working of the bitwise AND operator:

Snippet

```
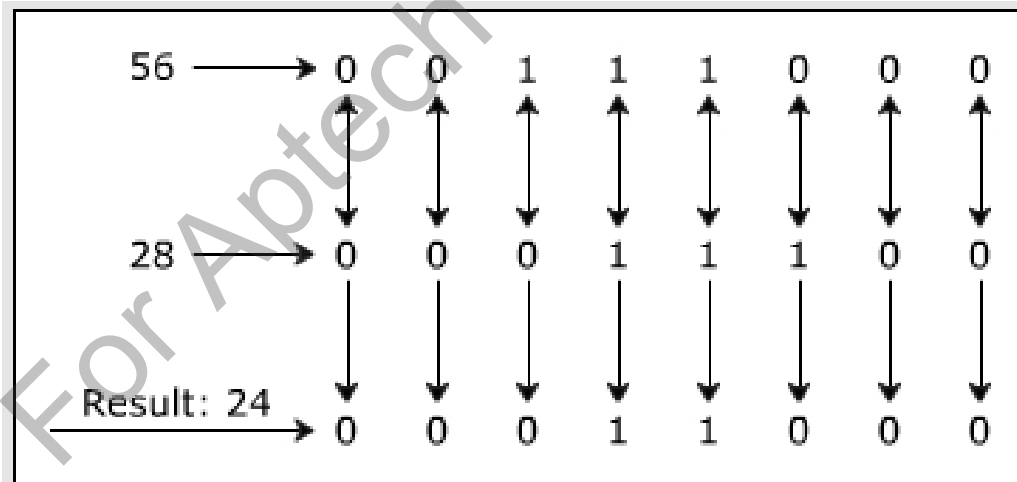result = 56 & 28; //(56 = 00111000 and 28 = 00011100)
Console.WriteLine(result);
```

- In the code:

  - The bitwise AND operator compares the corresponding bits of the two operands.

  - It returns 1 if both the bits in that position are 1 or else returns 0.

  - This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number.

  - This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is "24".

- The following figure displays the bitwise logical operators:

- The following code explains the working of the bitwise inclusive OR operator:

Snippet
```
result = 56 | 28;
Console.WriteLine(result);
```

- In the code:
  - The bitwise inclusive OR operator compares the corresponding bits of the two operands.
  - It returns 1 if either of the bits in that position is 1, else it returns 0.
  - This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number.
  - This number is automatically converted to integer, which is displayed as the output and the resultant output for the code is 60.

- The following code explains the working of the bitwise exclusive OR operator:

Snippet
```
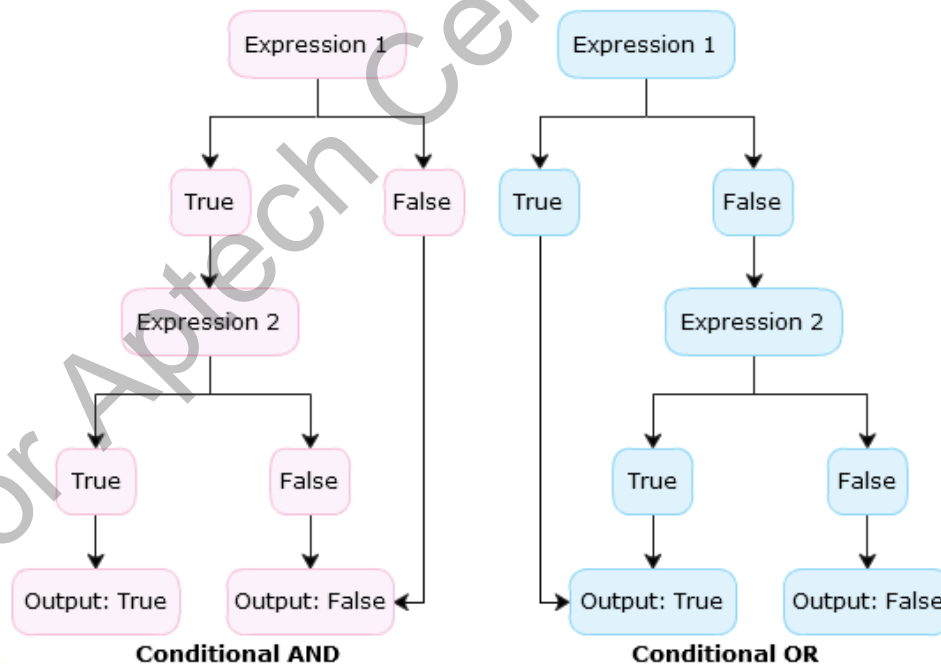result = 56 ^ 28;
Console.WriteLine(result);
```

- In the code:
  - The bitwise exclusive OR operator compares the corresponding bits of the two operands.
  - It returns 1 if only 1 of the bits in that position is 1, else it returns 0.
  - This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number.
  - This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is 36.

- There are two types of conditional operators, conditional AND (&&) and conditional OR (||).

- Conditional operators are similar to the boolean logical operators but have the following differences:
  - The conditional AND operator evaluates the second expression only if the first expression returns true because this operator returns true only if both expressions are true.
  - The conditional OR operator evaluates the second expression only if the first expression returns false because this operator returns true if either of the expressions is true.

Conditional AND

Conditional OR

◆ The following code displays the use of the conditional AND (&&) operator:

Snippet

```
int num = 0;
if (num >= 1 && num <= 10)
{
    Console.WriteLine("The number exists between 1 and 10");
}
else
{
    Console.WriteLine("The number does not exist between 1 and 10");
}
```

◆ In the code:

- The conditional AND operator checks the first expression.
- The first expression returns false, therefore, the operator does not check the second expression.
- The whole expression evaluates to false, the statement in the `else` block is executed.

Output

```
The number does not exist between 1 and 10
```

◆ The following code displays the use of the conditional OR (||) operator:

Snippet

```
int num = -5;
if (num < 0 || num > 10)
{
  Console.WriteLine("The number does not exist between 1 and 10");
}
else
{
  Console.WriteLine("The number exists between 1 and 10");
}
```

Output

```
The number does not exist between 1 and 10.
```

◆ In the code:

 ◈ The conditional OR operator checks the first expression.

 ◈ The first expression returns true, therefore, the operator does not check the second expression.

 ◈ The whole expression evaluates to true, the statement in the if block is executed.

- Two of the most common calculations performed in programming are increasing and decreasing the value of the variable by 1.

- In C#, the increment operator (++) is used to increase the value by 1 while the decrement operator (--) is used to decrease the value by 1.

- If the operator is placed before the operand, the expression is called pre-increment or pre-decrement.

- If the operator is placed after the operand, the expression is called post-increment or post-decrement.

- The following table depicts the use of increment and decrement operators assuming the value of the variable `valueOne` is 5:

Expression	Type	Result
valueTwo = ++ValueOne;	Pre-Increment	valueTwo = 6
valueTwo = valueOne++;	Post-Increment	valueTwo = 5
valueTwo = --valueOne;	Pre-Decrement	valueTwo = 4
valueTwo = valueOne--;	Post-Decrement	valueTwo = 5

◆ Assignment operators are used to assign the value of the right side operand to the operand on the left side using the equal to operator (=).

◆ The assignment operators are divided into two categories in C#. These are as follows:

   ◈ **Simple assignment operators**: The simple assignment operator is =, which is used to assign a value or result of an expression to a variable.

   ◈ **Compound assignment operators**: The compound assignment operators are formed by combining the simple assignment operator with the arithmetic operators.

◆ The following table shows the use of assignment operators assuming the value of the variable `valueOne` is 10:

Expression	Description	Result
valueOne += 5;	valueOne = valueOne + 5	valueOne = 15
valueOne -= 5;	valueOne = valueOne - 5	valueOne = 5
valueOne *= 5;	valueOne = valueOne * 5	valueOne = 50
valueOne %= 5;	valueOne = valueOne % 5	valueOne = 0

◆ The following code demonstrates how to use assignment operators.

Snippet

```
int valueOne = 5;
int valueTwo = 10;
Console.WriteLine("Value1 =" + valueOne);
valueOne += 4;
Console.WriteLine("Value1 += 4= " + valueOne);
valueOne -= 8;
Console.WriteLine("Value1 -= 8= " + valueOne);
valueOne *= 7;
Console.WriteLine("Value1 *= 7= " + valueOne);
valueOne /= 2;
Console.WriteLine("Value1 /= 2= " + valueOne);
Console.WriteLine("Value1 == Value2: {0}", (valueOne == valueTwo));
```

Output

```
Value1 =5

Value1 += 4= 9

Value1 -= 8= 1

Value1 *= 7= 7

Value1 /= 2= 3

Value1 == Value2: False
```

- Operators in C# have certain associated priority levels.
- The C# compiler executes operators in the sequence defined by the priority level of the operators.

Example

- The multiplication operator (*) has higher priority over the addition (+) operator. Thus, if an expression involves both the operators, the multiplication operation is carried out before the addition operation. In addition, the execution of the expression (associativity) is either from left to right or vice-versa depending upon the operators used.
- The following table lists the precedence of the operators, from the highest to the lowest precedence, their description and their associativity.

Precedence (where 1 is the highest)	Operator	Description	Associativity
1	()	Parentheses	Left to Right
2	++ or --	Increment or Decrement	Right to Left
3	*, /, %	Multiplication, Division, Modulus	Left to Right
4	+, -	Addition, Subtraction	Left to Right
5	<, <=, >, >=	Less than, Less than or equal to, Greater than, Greater than or equal to	Left to Right
6	=, !=	Equal to, Not Equal to	Left to Right
7	&&	Conditional AND	Left to Right
8	||	Conditional OR	Left to Right
9	=, +=, -=, *=, /=, %=	Assignment Operators	Right to Left

◆ The following code demonstrates precedence of operators:

Snippet

```
int valueOne = 10;
Console.WriteLine((4 * 5 - 3 ) / 6 + 7 - 8 % 5);
Console.WriteLine((32 < 4) || (8 == 8));
Console.WriteLine(((valueOne *= 6) > (valueOne += 5)) && ((valueOne /= 2) !=
(valueOne -= 5)));
```

◆ In the code:

- The variable **valueOne** is initialized to the value 10.

- The next three statements display the results of the expressions.

- The expression given in the parentheses is solved first.

Output
```
True
False
```

◆ The shift operators allow the programmer to perform shifting operations. The two shifting operators are as follows:

The Left Shift (<<) Operators	• The left shift operator allows shifting the bit positions towards the left side where the last bit is truncated and zero is added on the right.
The Right Shift (>>) Operators	• The right shift operator allows shifting the bit positions towards the right side and the zero is added on the left.

◆ The following code demonstrates the use of shift operators.

Snippet

```
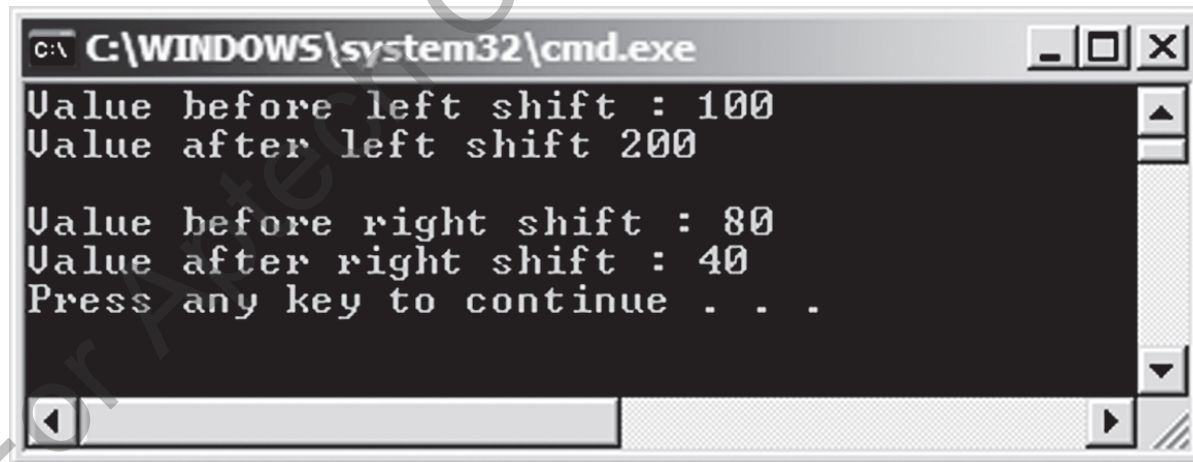using System;

class ShiftOperator
{
    static void Main(string[] args)
    {
        uint num = 100; // 01100100 = 100
        uint result = num << 1; // 11001000 = 200
        Console.WriteLine("Value before left shift : " + num);
        Console.WriteLine("Value after left shift " + result);
        num = 80; // 10100000 = 80
        result = num >> 1; // 01010000 = 40
        Console.WriteLine("\nValue before right shift : " + num);
        Console.WriteLine("Value after right shift : " + result);
    }
}
```

- In the code:

  - The `Main()` method of the class **ShiftOperator** performs the shifting operations.

  - The `Main()` method initializes the variable **num** to 100.

  - After shifting towards left, the value of **num** is doubled. Now, the variable **num** is initialized to 80.

  - After shifting towards right, the value of **num** is halved.

- Following is the output of code:

```
C:\WINDOWS\system32\cmd.exe                       _ □ ×

Value before left shift : 100
Value after left shift 200

Value before right shift : 80
Value after right shift : 40
Press any key to continue . . .
```

- The arithmetic operator (+) allows the programmer to add numerical values.

- However, if one or more operands are characters or binary strings, columns, or a combination of strings and column names into one expression, then the string concatenation operator concatenates them. In other words, the arithmetic operator (+) is overloaded for string values.

- The following code demonstrates the use of string concatenation operator.

Snippet

```
using System;
class Concatenation
{
    static void Main(string[] args)
    {
        int num = 6;
        string msg = "";
        if (num < 0)
        {
            msg = "The number " + num + " is negative";
        }
        else if ((num % 2) == 0)
        {
            msg = "The number " + num + " is even";
        }
        else
        {
            msg = "The number " + num + " is odd";
        }
        if(msg != "")
            Console.WriteLine(msg);
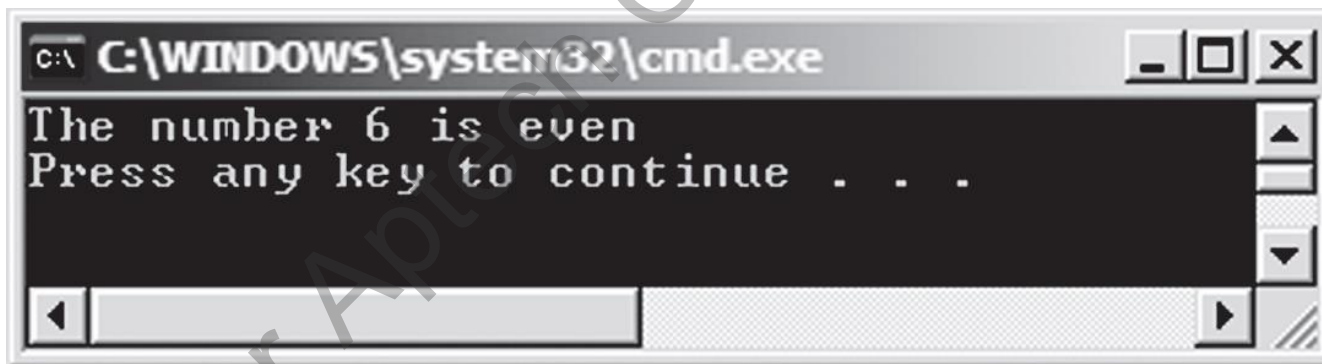
    }
}
```

- In the code:
  - The `Main()` method of the class **Concatenation** uses the construct to check whether a number is even, odd, or negative.
  - Depending upon the condition, the code displays the output by using the string concatenation operator (+) to concatenate the strings with numbers.

- The output shows the use of string concatenation operator.


```
C:\WINDOWS\system32\cmd.exe

The number 6 is even
Press any key to continue . . .
```

◆ The ?: is referred to as the conditional operator. It is generally used to replace the `if-else` constructs.

◆ Since it requires three operands, it is also referred to as the ternary operator.

◆ The first expression returns a `bool` value, and depending on the value returned by the first expression, the second or third expression is evaluated.

◆ If the first expression returns a true value, the second expression is evaluated, whereas if the first expression returns a false value, the third expression is evaluated.

Syntax

```
<Expression 1> ? <Expression 2>: <Expression 3>;
```

◆ where,

  ◈ `Expression 1`: Is a bool expression.

  ◈ `Expression 2`: Is evaluated if expression 1 returns a true value.

  ◈ `Expression 3`: Is evaluated if expression 1 returns a false value.

◆ The following code demonstrates the use of the ternary operator.

Snippet

```
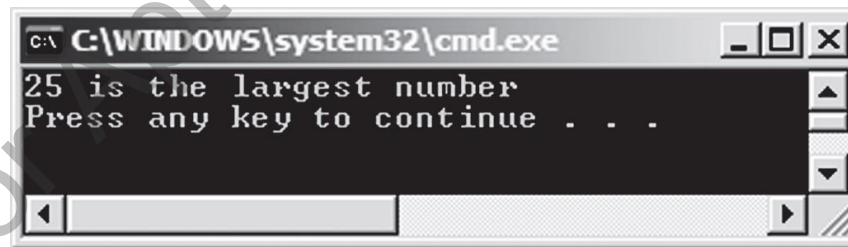using System;
class LargestNumber
{
    public static void Main()
    {
        int numOne = 5;
        int numTwo = 25;
        int numThree = 15;
        int result = 0;
        if (numOne > numTwo)
        {
            result = (numOne > numThree) ? result =
            numOne :
            result = numThree;
        }
        else
        {
            result = (numTwo > numThree) ? result =
            numTwo : result = numThree;
        }
        if(result != 0)
        Console.WriteLine("{0} is the largest number",

        result);

    }

}
```

- In the code:

  - The Main() method of the class **LargestNumber** checks and displays the largest of three numbers, **numOne, numTwo,** and **numThree.**

  - This largest number is stored in the variable **result.**

  - If **numOne** is greater than numTwo, then the ternary operator within the if loop is executed.

  - The ternary operator (?:) checks whether **numOne** is greater than **numThree.** If this condition is true, then the second expression of the ternary operator is executed, which will assign **numOne** to **result.**

  - Otherwise, if **numOne** is not greater than **numThree,** then the third expression of the ternary operator is executed, which will assign **numThree** to **result.**

  - Similar operation is done for comparison of **numOne** and **numTwo** and the **result** variable will contain the larger value out of these two.

- The following figure shows the output of the example using ternary operator:

- Data conversions is performed in C# through casting, a mechanism to convert one data type to another.

Example

- Consider the payroll system of an organization.

- The gross salary of an employee is calculated and stored in a variable of `float` type.

- The payroll department wants the salary amount as a whole number and thus, wants any digits after the decimal point of the calculated salary to be ignored.

- The programmer can achieve this using the typecasting feature of C#, which allows you to change the data type of a variable.

- C# supports two types of casting, namely Implicit and Explicit.

- Typecasting is mainly used to:
    - Convert a data type to another data type that belongs to the same or a different hierarchy.
    - Display the exact numeric output. For example, you can display exact quotients during mathematical divisions.
    - Prevent loss of numeric data if the resultant value exceeds the range of its variable's data type.

- Implicit typecasting refers to an automatic conversion of data types. This is done by the C# compiler.

- Implicit typecasting is done only when the destination and source data types belong to the same hierarchy.

- In addition, the destination data type must hold a larger range of values than the source data type.

- Implicit conversion prevents the loss of data as the destination data type is always larger than the source data type.

- For example, if you have a value of `int` type, you can assign that value to the variable of `long` type.

- The following code shows an example of implicit conversion.

Snippet	

	`int valueOne = 34;`
	`float valueTwo;`
	
	`valueTwo = valueOne;`

- In the code:
  - The compiler generates code that automatically converts the value in **valueOne** into a floating-point value before storing the result in **valueTwo**. Converting an integer value to a floating point value is safe.

- Implicit typecasting is carried out automatically by the compiler.

- The C# compiler automatically converts a lower precision data type into a higher precision data type when the target variable is of a higher precision than the source variable.

- The following figure illustrates the data types of higher precision to which they can be converted:

- Explicit typecasting refers to changing a data type of higher precision into a data type of lower precision.

- For example, using explicit typecasting, you can manually convert the value of `float` type into `int` type.

- This typecasting might result in loss of data because when you convert the `float` data type into the `int` data type, the digits after the decimal point are lost.

- The following is the syntax for explicit conversion:

Syntax

```
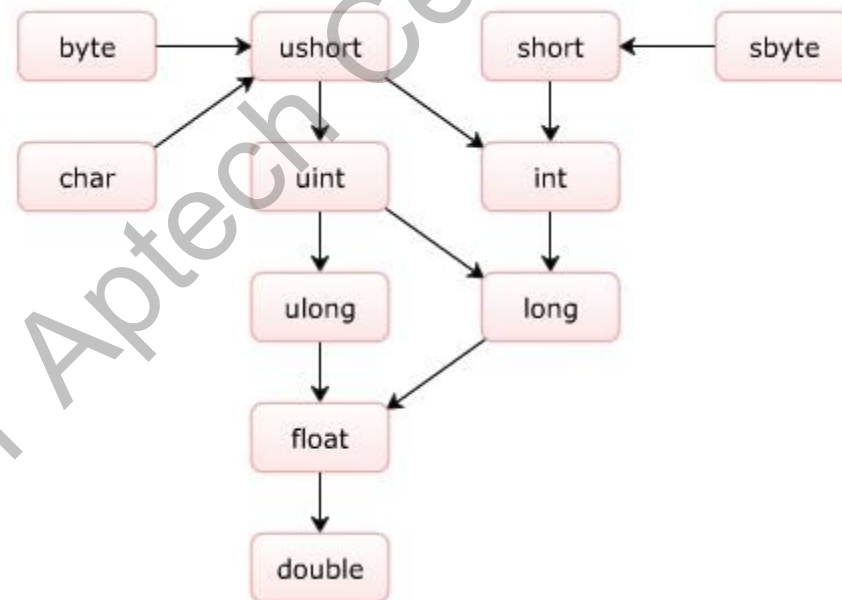<target data type><variable name> = (target data
type)<source data type>;
```

where,

`target data type`: Is the resultant data type.
`variable name`: Is the name of the variable, which is of the target data type.
`target data type`: Is the target data type in parentheses.
`source data type`: Is the data type which is to be converted.

◆ The following code displays the use of explicit conversion for calculating the area of a square:

Snippet

```
double side = 10.5;
int area;
area = (int)(side * side);

Console.WriteLine("Area of the square = {0}", area);
```

Output

```
Area of the square = 110
```

◆ There are two ways to implement explicit typecasting in C# using the built-in methods:

◈ **Using `System.Convert` class**: This class provides useful methods to convert any built-in data type to another built-in data type.

◈ **Using `ToString()` method**: This method belongs to the `Object` class and converts any data type value into `string`.

◆ The code displays a `float` value as `string` using the `ToString()` method:

Snippet

```
float flotNum = 500.25F;
string stNum = flotNum.ToString();
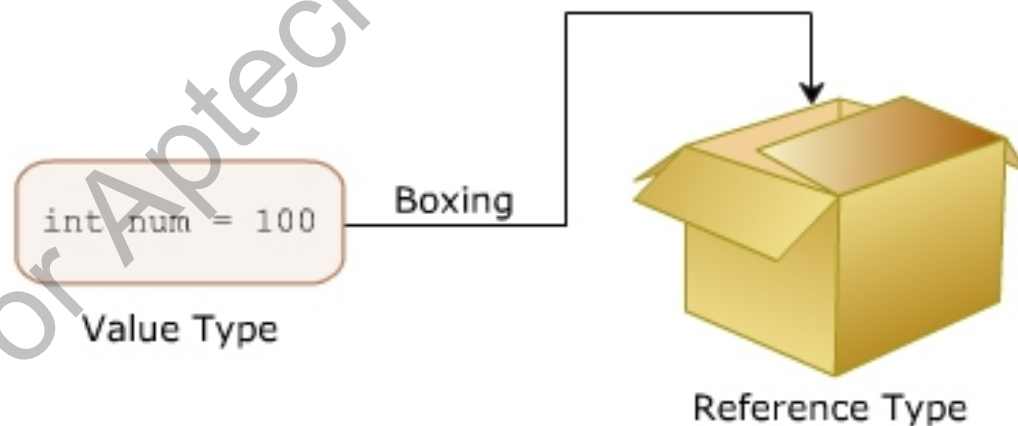Console.WriteLine(stNum);
```

◆ In the code:

◈ The value of `float` type is converted to string type using the `ToString()` method. The string is displayed as output in the console window.

Output   `500.25`

- Boxing is a process for converting a value type, like integers, to its reference type, like `objects` that is useful to reduce the overhead on the system during execution because all value types are implicitly of `object` type.

- To implement boxing, you need to assign the value type to an `object`.

- While boxing, the variable of type object holds the value of the value type variable which means that the object type has the copy of the value type instead of its reference.

- Boxing is done implicitly when a value type is provided instead of the expected reference type.

- The figure illustrates with an analogy the concept of boxing:

- The syntax for boxing is as follows:

| Syntax | `object <instance of the object class> = <variable of value type>;` |

where,

- `object:` Is the base class for all value types.
- `instance of the object class:` Is the name referencing the `Object` class.
- `variable of value type:` Is the identifier whose data type is of value type.

- The following code demonstrates the use of implicit boxing:

Snippet	

```
int radius = 10;
double area;
area = 3.14 * radius * radius;
object boxed = area;

Console.WriteLine("Area of the circle = {0}",boxed);
```

- In the code:

- Implicit boxing occurs when the value of `double` variable, **area,** is assigned to an object, **boxed**.

| Output | `Area of the circle = 314` |

- The following code demonstrates the use of explicit boxing:

Snippet

```
float radius = 4.5F;
double circumference;
circumference = 2 * 3.14 * radius;
object boxed = (object)circumference;
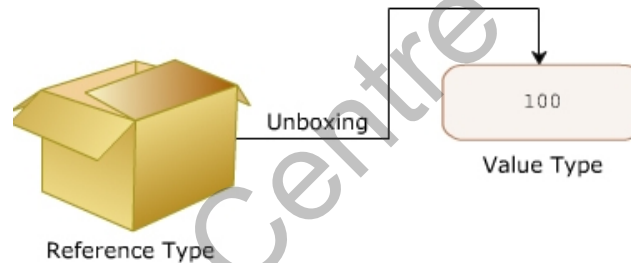Console.WriteLine("Circumference of the circle = {0}", circumference);
```

In the code:

Explicit boxing occurs by casting the variable, **circumference**, which is assigned to object, **boxed**.

Output

```
Circumference of the circle = 28.26
```

- Unboxing refers to converting a reference type to a value type.

- The stored value inside an object is unboxed to the value type.

- The object type must be of the destination value type. This is explicit conversion, without which the program will give an error.

- The following figure illustrates with an analogy the concept of unboxing:

- The syntax for unboxing is as follows:

```
<target value type><variable name> = (target value
    type) <object type>;
```

- where,
  - `target value type:` Is the resultant data type.
  - `variable name:` Is the name of the variable of value type.
  - `target value type:` Is the resultant value type in parentheses.
  - `object type:` Is the reference name of the Object class.

◆ The following code demonstrates the use of unboxing while calculating the area of the rectangle:

Snippet

```
int length = 10;
int breadth = 20;
int area;
area = length * breadth;
object boxed = area;
int num = (int)boxed;

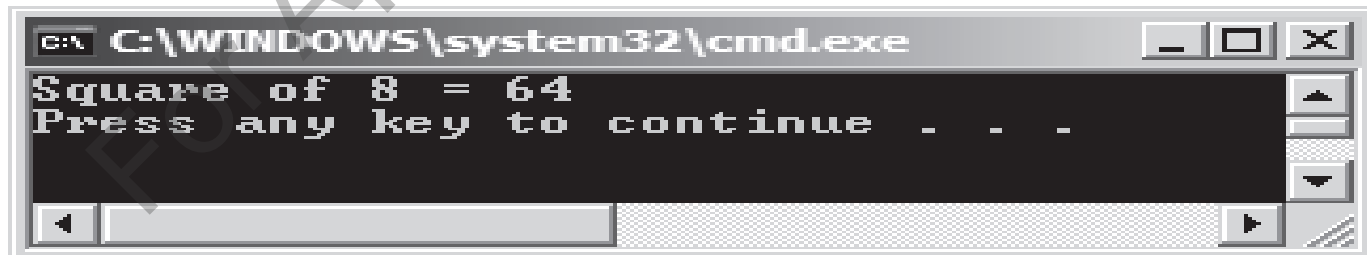Console.WriteLine("Area of the rectangle= {0}", num);
```

◆ In the code:

  ◈ Boxing is done when the value of the variable **area** is assigned to an object, **boxed.** However, when the value of the object, boxed, is to be assigned to **num**, unboxing is done explicitly because **boxed** is a reference type and **num** is a value type.

◆ The following code demonstrates how boxing and unboxing occur:

Snippet

```
using System;
class Number
{
        static void Main(string[] args)
        {
              int num = 8;
              int result;
              result = Square(num);
              Console.WriteLine("Square of {0} = {1}", num, result);
        }
        static int Square(object inum)
        {
              return (int)inum * (int)inum;
        }

}
```

◆ In the code:

- The `Main()` method calculates the square of the specified value.

- Boxing occurs when the variable **num,** which is of value type, is passed to the **Square()** method whose input parameter, **inum,** is of reference type.

- Unboxing occurs when this reference type variable, **inum**, is converted to value type, int, before its square is returned to variable **result**.

◆ The following figure shows the output of the code:

```
C:\WINDOWS\system32\cmd.exe
Square of 8 = 64
Press any key to continue . . .
```

- Statements are executable lines of code that build up a program.

- Expressions are a part of statements that always result in generating a value as the output.

- Operators are symbols used to perform mathematical and logical calculations.

- Each operator in C# is associated with a priority level in comparison with other operators.

- You can convert a data type into another data type implicitly or explicitly in C#.

- A value type can be converted to a reference type using the boxing technique.

- A reference type can be converted to a value type using the unboxing technique.