Session: 3

Statements and Operators

Objectives

- Define and describe statements and expressions
- Explain the types of operators
- Explain the process of performing data conversions in C#

Types of Statements

- Similar to statements in C and C++, the C# statements are classified into seven categories:
  - Selection Statements
  - Iteration Statements
  - Jump Statements
  - Exception Handling Statements
  - Fixed Statement
  - Lock Statement
  - Checked and Unchecked Statements

Checked and Unchecked Statements

- 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
```

```
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);
        }
    }

}
```

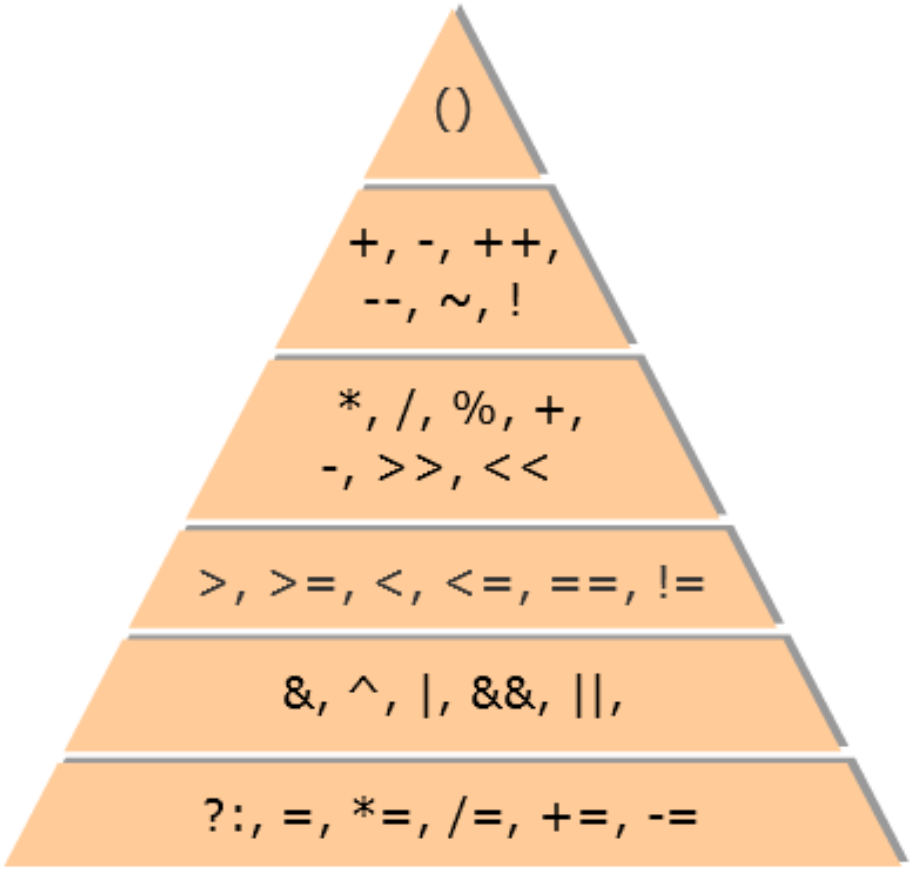
Expressions – Definition

- Expressions 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.

Snippet

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

- These are classified into six categories :
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Conditional Operators
  - Increment and Decrement Oper
  - Assignment Operators



Relational Operators

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

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`

- **Boolean Logical Operators**:
  - perform boolean logical operations on both the operands.
  - return a boolean value based on the logical operator used.

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')`

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

- **Bitwise Logical Operators:**
  - perform logical operations on the corresponding individual bits of two operands.

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 :

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

Increment and Decrement Operators

- 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.
- Example with 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 – Types

- are used to assign the value of the right side operand to the operand on the left side using the equal to operator (=).
- are divided into two categories. These are as follows:
  - **Simple assignment operators**
  - **Compound assignment operators**
- Example with **`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

String Concatenation Operator

- if one or more operands of arithmetic operator (+) are characters or binary strings … then the string concatenation operator

```
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);
      }
}
```

Ternary or Conditional Operator

- 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.
- If the first expression returns a true value, the second expression is evaluated, else, the third expression is evaluated.

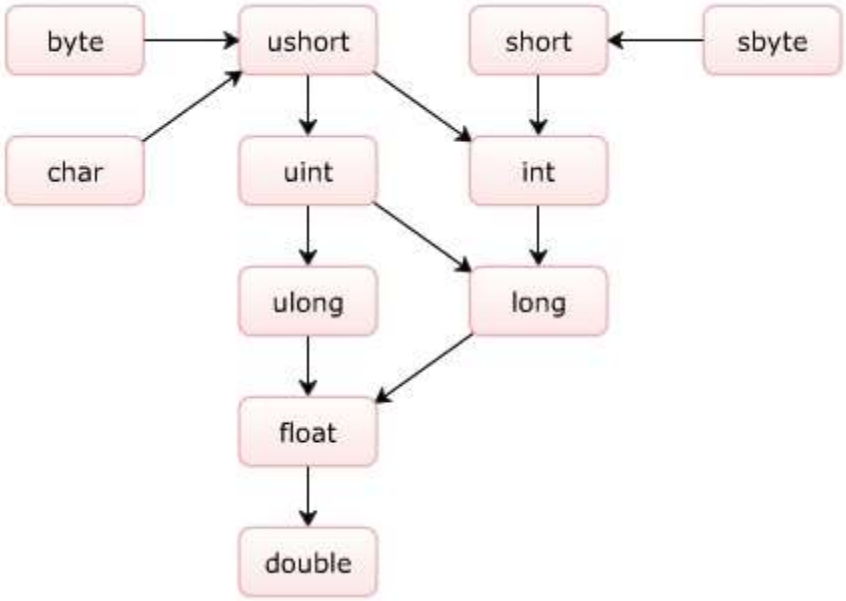
Syntax

```
<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.

Implicit Conversions for C# Data Types – Rules

- 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 Type Conversion – Definition

- 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
```

Explicit Type Conversion – Implementation

- There are two ways to implement explicit typecasting :
  - **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

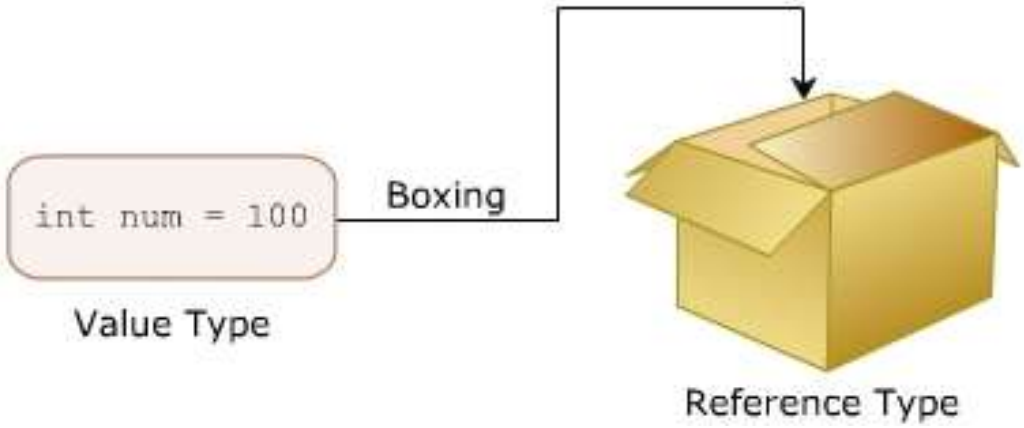
```
float f = 500.25F;
string stNum = f.ToString();
Console.WriteLine(stNum);
```

Output

```
500.25
```

Boxing and Unboxing

- 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:



Summary

- 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.