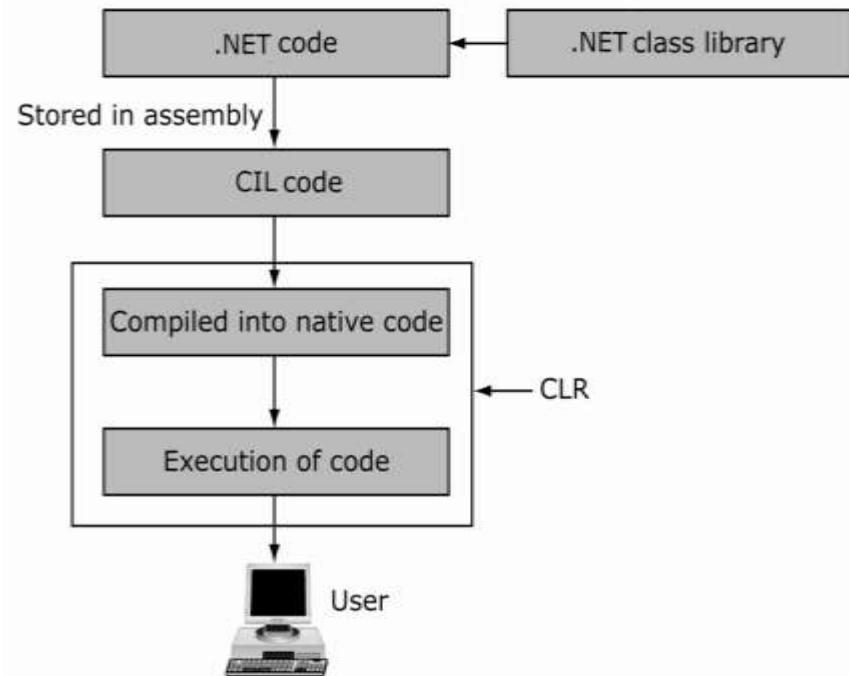


Session: 1

# Building Applications Using C#

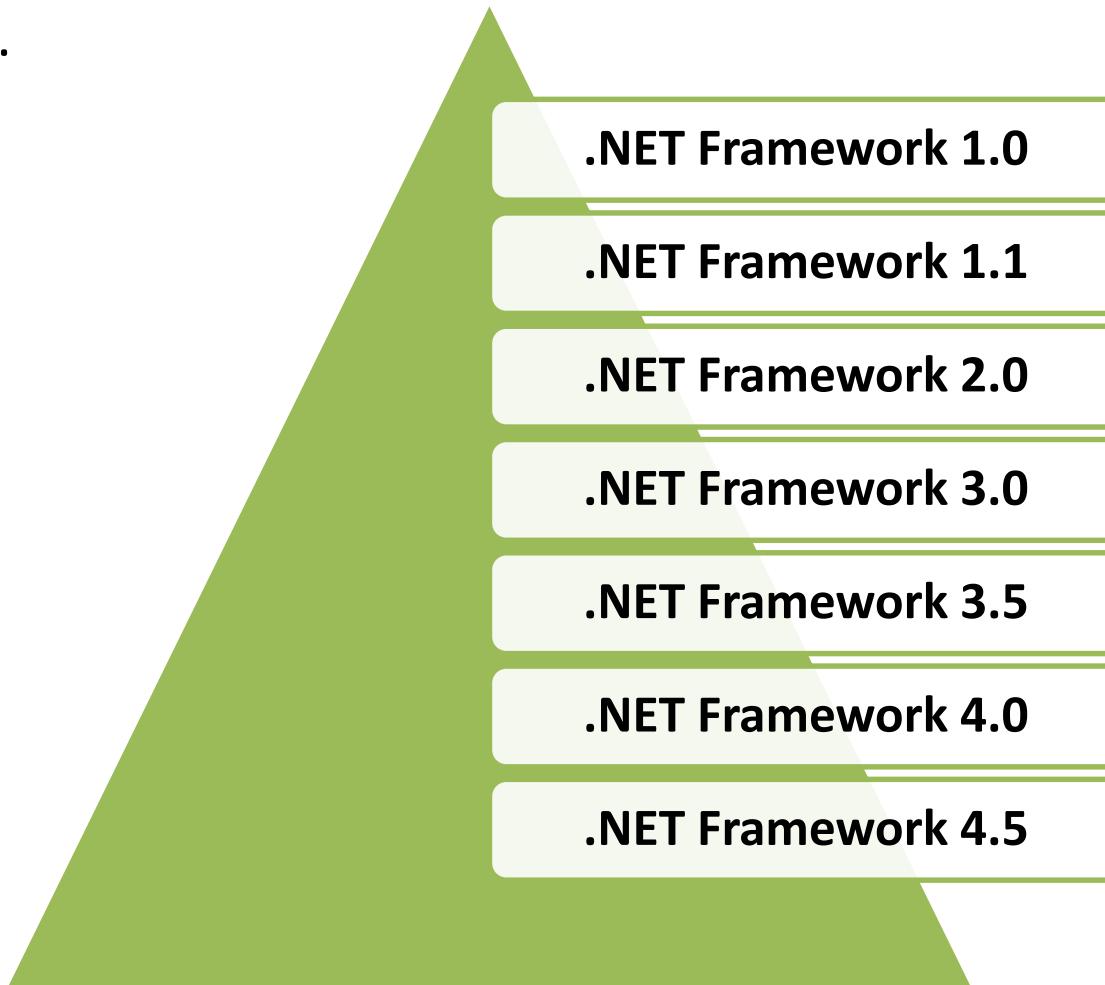
- ◆ Define and describe the .NET Framework
- ◆ Explain the C# language features
- ◆ Define and describe the Visual Studio 2012 environment
- ◆ Explain the elements of Microsoft Visual Studio 2012 IDE

- ◆ In traditional Windows applications:
  - ❖ Codes were directly compiled into the executable native code of the operating system.
- ◆ With .NET framework:
  - ❖ Code is compiled into CIL (formerly called MSIL) and stored in a file called assembly.
  - ❖ Assembly is then compiled to the native code by CLR



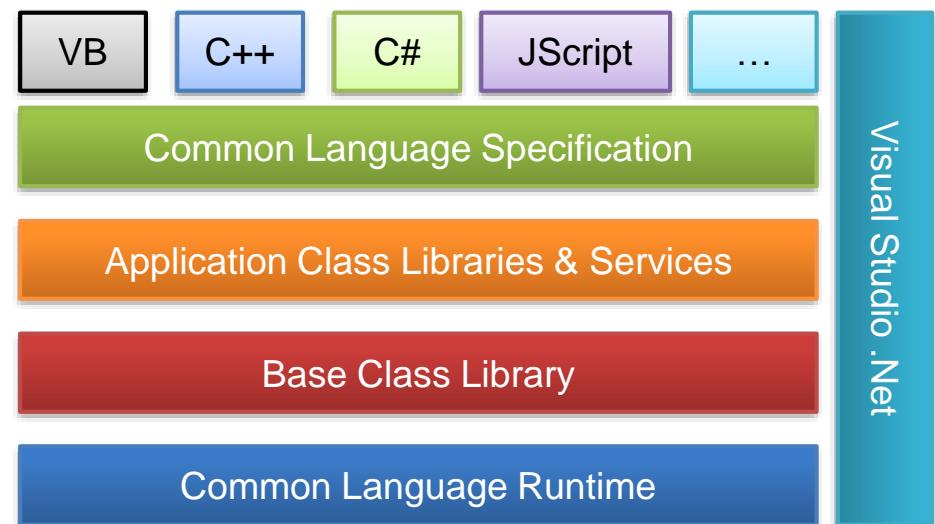
# The .NET Framework Architecture

- ◆ Microsoft has released different versions of the .NET Framework including additional capabilities and functionalities with every newer version.

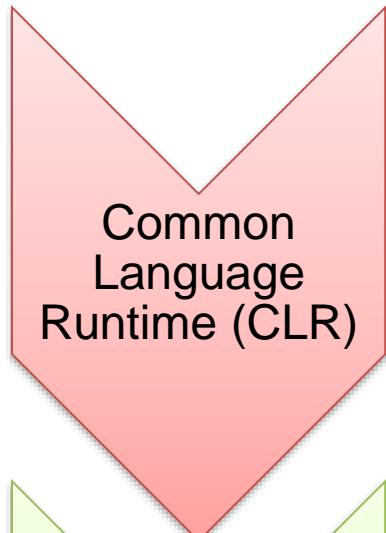


# The .NET Framework Fundamentals

- ◆ The .NET Framework is an essential component for building and running the next generation applications and XML Web services.
- ◆ It is designed to:
  - ◆ Provide consistent OOP environment.
  - ◆ Provide a code-execution environment that:
    - Minimizes software deployment and versioning conflicts
    - Promotes safe execution of code
  - ◆ Provide a consistent development experience across varying types of applications such as Windows apps and Web-apps.



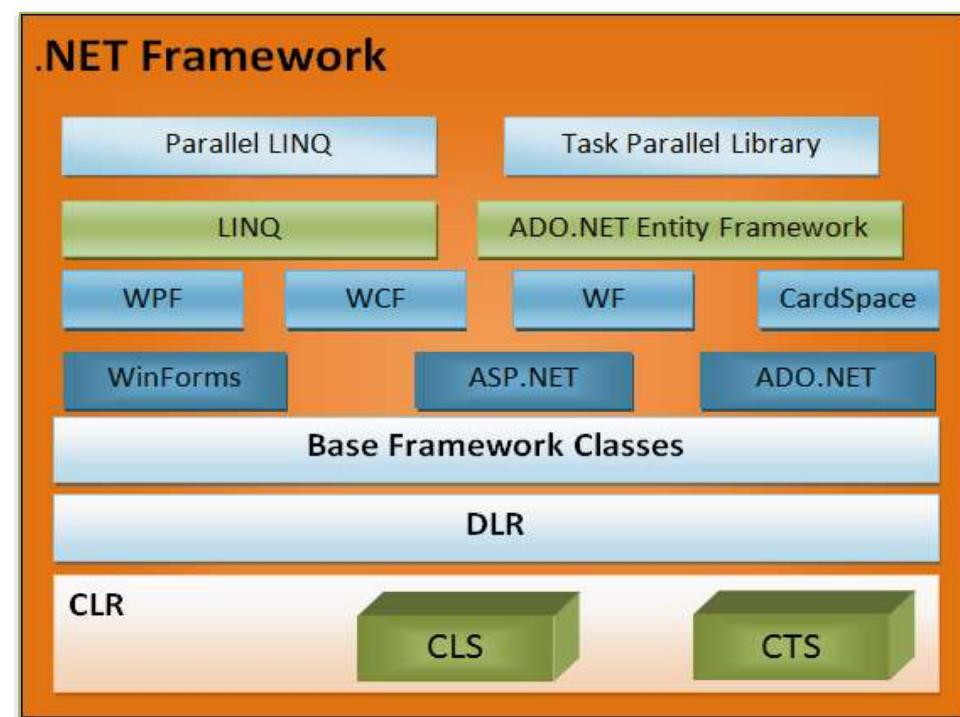
- ◆ The two core components of the .NET Framework are:



- Is a backbone of .NET Framework
  - Performs various functions such as:
    - Memory management, Code execution
    - Error handling, Code safety verification
    - Garbage collection
- 
- Is a comprehensive object-oriented collection of reusable types.
  - Used to develop applications ranging from traditional command-line to GUI applications that can be used on the Web.

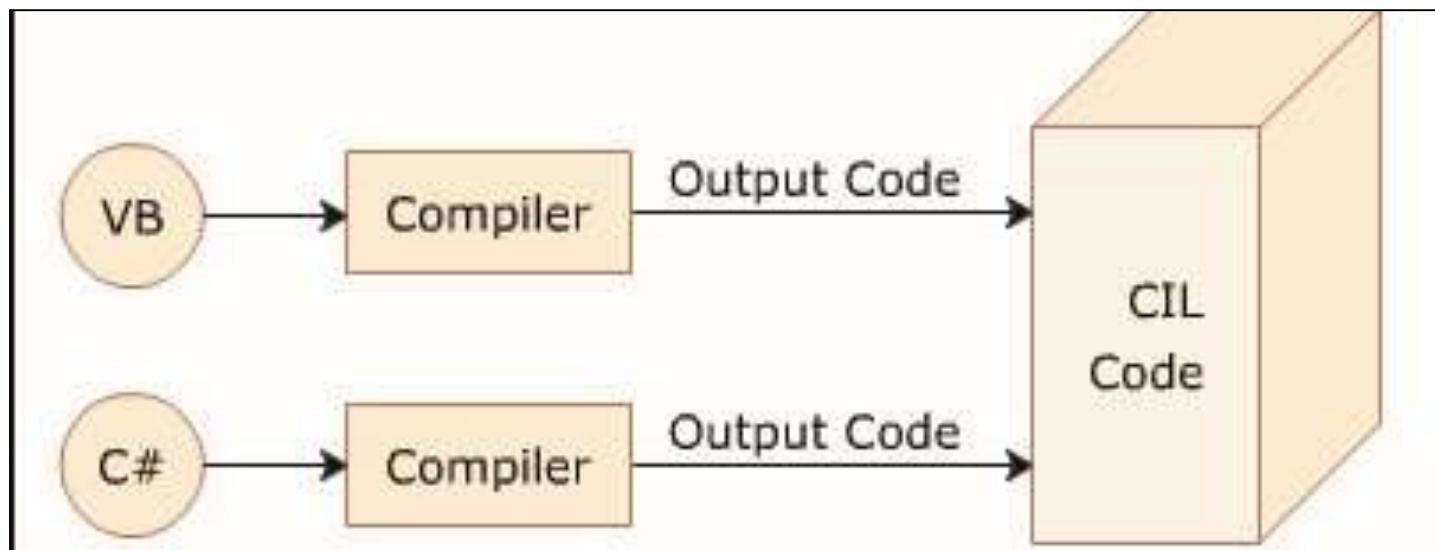
# Other Components of .NET Framework

- ◆ Following are some other important components:
  - ❖ Common Language Specification (CLS)
  - ❖ Common Type System (CTS)
  - ❖ Base Framework Classes
  - ❖ ASP.NET
  - ❖ ADO.NET
  - ❖ WPF
  - ❖ WCF
  - ❖ LINQ
  - ❖ ADO.NET Entity Framework
  - ❖ Parallel LINQ
  - ❖ Task Parallel Library



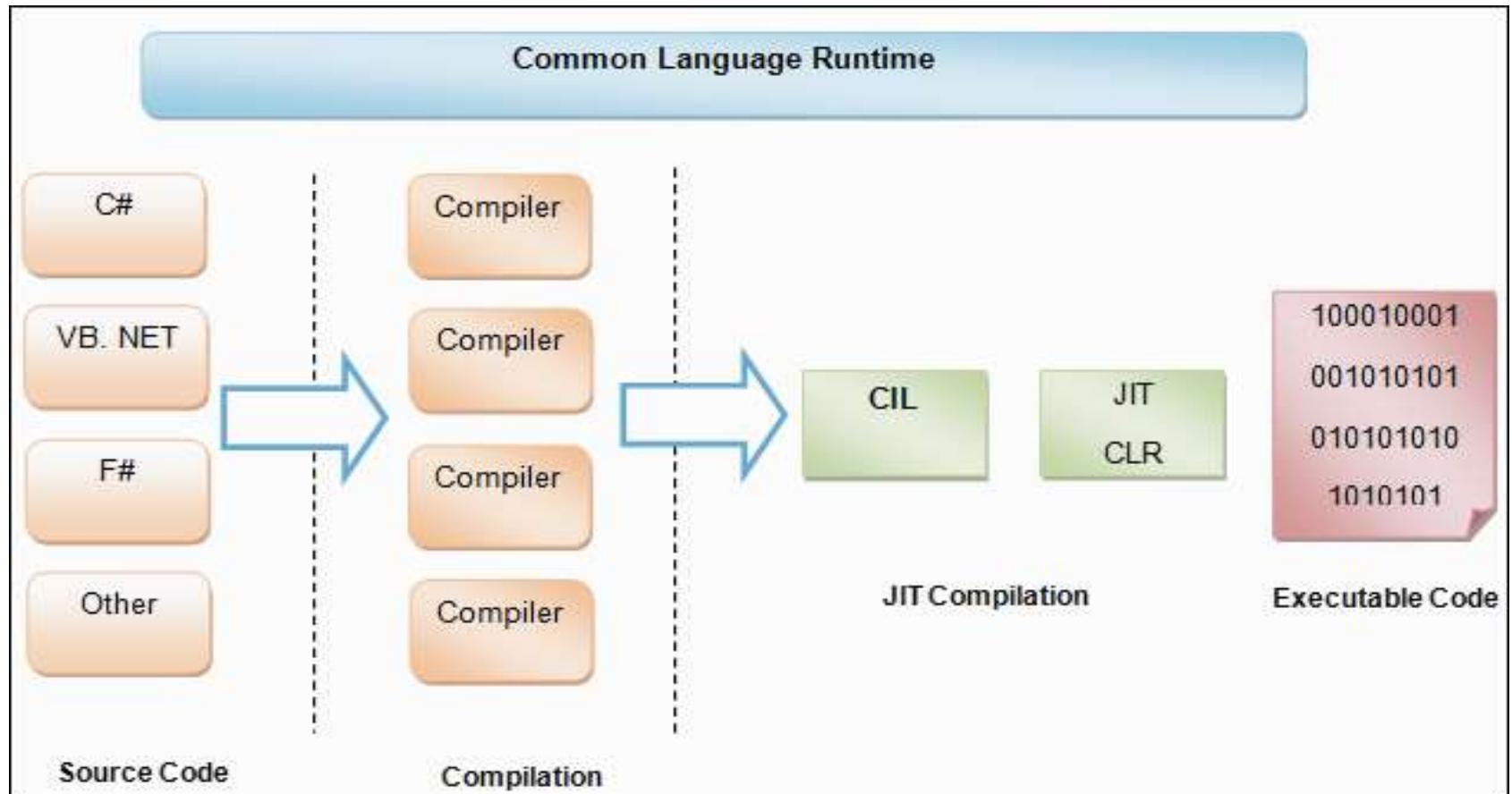
# Common Intermediate Language (CIL)

- The following figure depicts the concept of CIL:



# Common Language Runtime (CLR)

- The following figure shows a more detailed look at the working of the CLR:



## VS Professional 2012

- This is the entry-level edition that provides support for developing and debugging applications, such as Web, desktop, cloud-based, and mobile applications.

## Professional with MSDN

- provides all the features of the VS Professional 2012 along with an MSDN subscription.
- includes Team Foundation Server and provides access to cloud, Windows Store, and Windows Phone Marketplace.

## Test Professional with MSDN

- targets testers and Quality Assurance (QA) professionals by providing project management tools, testing tools, and virtual environment to perform application testing.

## Premium with MSDN

- provides all the features of the combined VS Professional 2012 and VS Test Professional 2012 with MSDN editions.
- supports peer code review, UI validation through automated tests, and code coverage analysis to determine the amount of code being tested.

## Ultimate with MSDN

- has all the features of the other editions
- supports designing architectural layer diagrams, performing Web performance and load testing, and analyzing diagnostic data collected from runtime systems.

- ◆ Visual Studio 2012 supports multiple programming languages such as:
  - ◆ Visual Basic .NET
  - ◆ Visual C++
  - ◆ Visual C#
  - ◆ Visual J#
- ◆ The **classes and libraries** used in the VS 2012 IDE are common for all the languages.
- ◆ It makes Visual Studio 2012 more flexible.

- ◆ Console applications that are created in C# - run in a console window. This window provides simple text-based output.
- ◆ The **csc** (C Sharp Compiler) command can be used to compile a C# program.
- ◆ Following are the steps to compile and execute a program:
  1. **Create a New Project.**
  2. **Compile a C# Program.**
  3. **Execute the Program.**

- ◆ **Compile a C# Program:**

- ◆ A C# program can be compiled using the following syntax:

```
csc <file.cs>
```

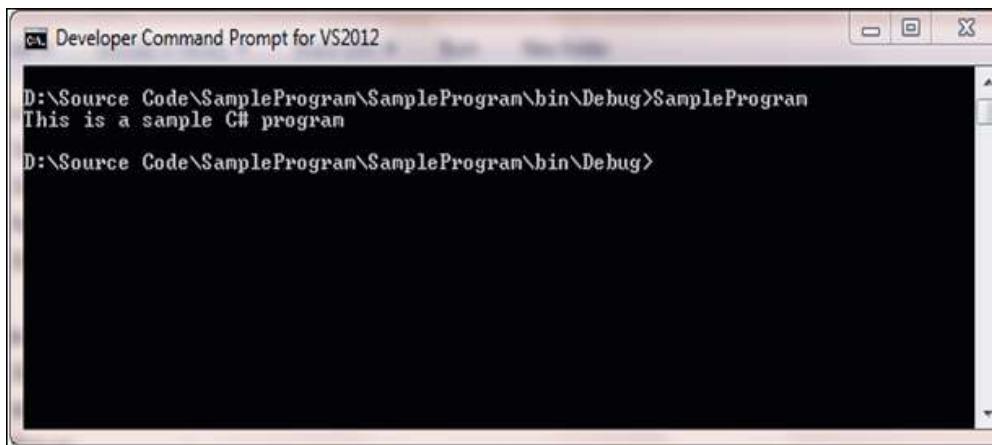
## Example

```
csc SampleProgram.cs
```

- ◆ In the example:
    - **SampleProgram**: Specifies the name of the program to be compiled.
    - This command generates an executable file **SampleProgram.exe**.

- ◆ Execute the Program:

- ◆ Open the Developer Command Prompt for VS2012, and browse to the directory that contains the **.exe** file.
- ◆ Type the file name at the command prompt.
- ◆ The following figure shows the developer command prompt for VS2012 window:



- ◆ The .exe file is known as **portable EXE** as it contains machine-independent instructions.
- ◆ The portable EXE works on any operating system that supports the .NET platform.

- ◆ The IDE also provides the necessary support to compile and execute C# programs.
- ◆ Following are the steps to compile and execute C# programs:
  - ❖ **Compiling the C# Program**
    - Select **Build <application name>** from the **Build** menu. This action will create an executable file (**.exe**).
  - ❖ **Executing the Program:**
    - Select **Start Without Debugging** from the **Debug** menu.



- ◆ The .NET Framework is an infrastructure that enables building, deploying, and running different types of applications and services using .NET technologies.
- ◆ The two core components of the .NET Framework which are integral to any application or service development are the CLR and the .NET Framework class library.
- ◆ The CLR is a virtual machine component of .NET that is used to convert the CIL code to the machine language code.
- ◆ Visual Studio 2012 provides the environment to create, deploy, and run applications developed using the .NET framework.
- ◆ Some of the languages supported by Visual Studio 2012 include Visual Basic .NET, Visual C++, Visual C#, Visual J#, and Visual F#.
- ◆ C# is an object-oriented language derived from C and C++.

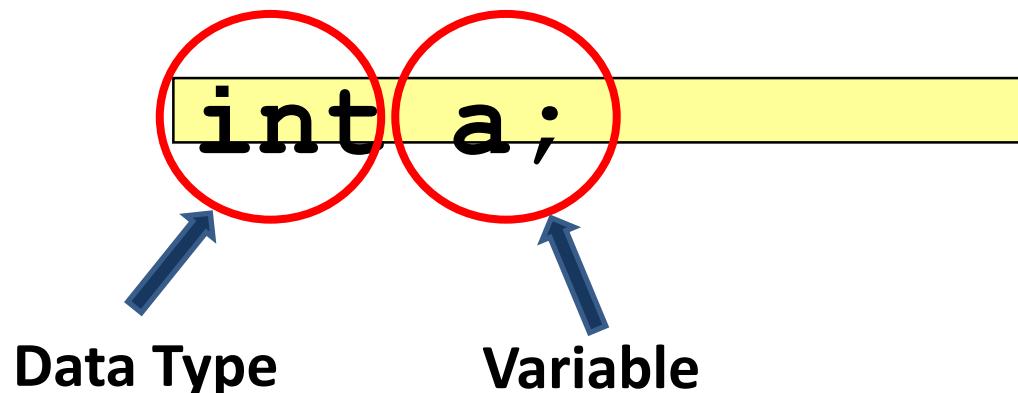
Session: 2

# Variables and Data Types

- ◆ Define and describe variables and data types in C#
- ◆ Explain comments and XML documentation
- ◆ Define and describe constants and literals
- ◆ List the keywords and escape sequences
- ◆ Explain input and output

- ◆ A variable is used to store data in a program and is declared with an associated data type.
- ◆ A variable has a name and may contain a value.
- ◆ Syntax:

```
data_type <variableName> [= <value>] ;
```



- ◆ In C# data types are divided into two categories:

## Value Types

- store actual values in a stack → faster memory allocation.
- Most of the built-in data types are value types such as: int, float, double, char, and bool.
- Include user-defined value types: struct and enum

## Reference Types

- store the memory address of variables in a stack, store actual values in a heap.
- These values can either belong to a built-in data type or a user-defined data type.

# Pre-defined Data Types

- ◆ Are basic data types that have a pre-defined range and size.

Data Type	Size	Range
<b>byte</b>	Unsigned 8-bit integer	0 to 255
<b>sbyte</b>	Signed 8-bit integer	-128 to 127
<b>short</b>	Signed 16-bit integer	-32,768 to 32,767
<b>ushort</b>	Unsigned 16-bit integer	0 to 65,535
<b>int</b>	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
<b>uint</b>	Unsigned 32-bit integer	0 to 4,294,967,295
<b>long</b>	Signed 64-bit integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>ulong</b>	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
<b>float</b>	32-bit floating point with 7 digits precision	$\pm 1.5e-45$ to $\pm 3.4e38$
<b>double</b>	64-bit floating point with 15-16 digits precision	$\pm 5.0e-324$ to $\pm 1.7e308$
<b>Decimal</b>	128-bit floating point with 28-29 digits precision	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$
<b>Char</b>	Unicode 16-bit character	U+0000 to U+ffff
<b>bool</b>	Stores either true or false	true or false

- ◆ Declared for value types rather than for reference types.
- ◆ Have to initialize at the time of its declaration.
- ◆ The following syntax is used to initialize a constant:

```
const <data type> <identifier name> = <value>;
```

```
const float _pi = 3.14F;  
float radius = 5;  
float area = _pi * radius * radius;  
Console.WriteLine("Area of the circle is " + area);
```

- ◆ **Boolean Literal:** has two values, `true` or `false`.  
For example, `bool val = true;`
- ◆ **Integer Literal:** can be assigned to `integer` data types.  
Suffixes for integer literals include U, L, UL, or LU. U denotes `uint` or `ulong`, L denotes `long`. UL and LU denote `ulong`.  
For example, `long val = 53L;`
- ◆ **Real Literal:** is assigned to `real` data types.  
Suffix letters include F denotes `float`, D denotes `double`, and M denotes `decimal`.  
For example, `float val = 1.66F;`

- ◆ **Character Literal** is assigned to a char data type. A character literal is always enclosed in single quotes.  
For example, `char val = 'A' ;`

- ◆ **String Literal:** There are two types of string literals in C#, regular and verbatim.

- regular string is a standard string.
- verbatim string is prefixed by the '@' character.

A string literal is always enclosed in double quotes.

For example, `string mailDomain = "gmail.com" ;`

- ◆ **Null Literal:** The null literal has only one value, null.

For example, `string email = null ;`

# Escape Sequence Characters in C#

Escape Sequence Characters	Non-Printing Characters
\'	Single quote, needed for character literals.
\\"	Double quote, needed for string literals.
\\"\\	Backslash, needed for string literals.
\0	Unicode character 0.
\a	Alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\v	Vertical tab.
\t	Horizontal tab.
\?	Literal question mark.
\ooo	Matches an ASCII character, using a three-digit octal character code.
\xhh	Matches an ASCII character, using hexadecimal representation (exactly two digits). For example, \x61 represents the character 'a'.
\uhhhh	Matches a Unicode character, using hexadecimal representation (exactly four digits). For example, the character \u0020 represents a space.

- ◆ **Single-line Comments:** Begin with two forward slashes (//).

## Snippet

```
// This block of code will add two numbers  
int doSum = 4 + 3;
```

- ◆ **Multi-line Comments:** Begin with a forward slash followed by an asterisk /\*) and end with an asterisk followed by a forward slash (\*/).

## Snippet

```
/* This is a block of code that will multiply two  
numbers, divide the resultant value by 2 and display  
the quotient */  
  
int doMult = 5 * 20;  
  
int doDiv = doMult / 2;  
  
Console.WriteLine("Quotient is:" + doDiv)
```

- ◆ In C#, all console operations are handled by the **Console** class of the **System** namespace.  
A namespace is a collection of classes having similar functionalities.
- ◆ There are two output methods that write to the standard output stream as follows:
  - ❖ **Console.WriteLine()**
  - ❖ **Console.ReadLine()**
- ◆ Two methods that read data from standard input stream :
  - ❖ **Console.Read()**
  - ❖ **Console.ReadLine()**

- ◆ **Convert** class in the **System** namespace is used to convert one base data type to another base data type.

```
Console.WriteLine("Enter your name: ");
string userName = Console.ReadLine();
Console.WriteLine("Enter your age: ");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Enter the salary: ");
double salary = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Name: {0}, Age: {1}, Salary: {2} ",
userName, age, salary);
```

## Output

```
Enter your name: David Blake
Enter your age: 34
Enter the salary: 3450.50
Name: David Blake, Age: 34, Salary: 3450.50
```

# Using Numeric Format Specifiers

Format Specifier	Name	Description
<b>C or c</b>	Currency	The number is converted to a string that represents a currency amount.
<b>D or d</b>	Decimal	The number is converted to a string of decimal digits (0-9), prefixed by a minus sign in case the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. This format is supported for fundamental types only.
<b>E or e</b>	Scientific (Exponential)	The number is converted to a string of the form '-d.ddd...E+ddd' or '-d.ddd...e+ddd', where each 'd' indicates a digit (0-9).

- The following code demonstrates the conversion of a numeric value using C, D, F and E format specifiers:

## Snippet

```
int d = 100;  
float f = 12.4566F;  
Console.WriteLine("Currency format = {0:C6}",d);  
Console.WriteLine("Decimal format ={0:D6}",d);  
Console.WriteLine("Floating format={0:F2}",d);  
Console.WriteLine("D = {0:E4}; F = {1,8:0.00}", d, f);
```

## Output

```
Currency format = $100.000000  
Decimal format = 000100  
Floating format= 100.00  
D = 1.0000E+002; f = 12.45
```

# Standard Date and Time Format Specifiers

- ◆ is special characters displaying the date and time values in different formats.
- ◆ Syntax :

```
Console.WriteLine("{format specifier}", <datetime object>);
```

```
DateTime dt = DateTime.Now;  
Console.WriteLine("Short date, time with seconds (G) :{0:G}", dt);  
Console.WriteLine("Month and day (m) :{0:m}", dt);  
Console.WriteLine("Short time (t) :{0:t}", dt);  
Console.WriteLine("Short time with seconds (T) :v{0:T}", dt);  
Console.WriteLine("Year and Month (y) :{0:y}", dt);
```

```
Short date, time with seconds (G): 6/23/2018 6:01:24 PM  
Month and day (m): June 23  
Short time (t): 6:01 PM  
Short time with seconds (T): 6:01:24 PM  
Year and Month (y): June 2018
```

# Custom DateTime Format Strings

Format	Description
ddd	abbreviated name of the day of the week
dddd	full name of the day of the week
FF	two digits of the seconds fraction
H	Hour, 0 to 23
HH	Hour, 00 to 23
MM	Month, 01 to 12
MMM	abbreviate name of month
s	Seconds, 0 to 59

```
Date is Sat Jun 23, 2018  
Time is 06:19 PM  
24 hour time is 18:19  
Time with seconds: 18:19:28 PM
```

```
DateTime date = DateTime.Now;  
  
Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", date);  
Console.WriteLine("Time is {0:hh:mm tt}", date);  
Console.WriteLine("24 hour time is {0:HH:mm}", date);  
Console.WriteLine("Time with seconds: {0:HH:mm:ss tt}", date);
```

- ◆ A variable is a named location in the computer's memory and stores values.
- ◆ Comments are used to provide detailed explanation about the various lines in a code.
- ◆ Constants are static values that you cannot change throughout the program execution.
- ◆ Keywords are special words pre-defined in C# and they cannot be used as variable names, method names, or class names.
- ◆ Escape sequences are special characters prefixed by a backslash that allow you to display non-printing characters.
- ◆ Console operations are tasks performed on the command line interface using executable commands.
- ◆ Format specifiers allow you to display customized output in the console window.

Session: 3

# Statements and Operators

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

- ◆ 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

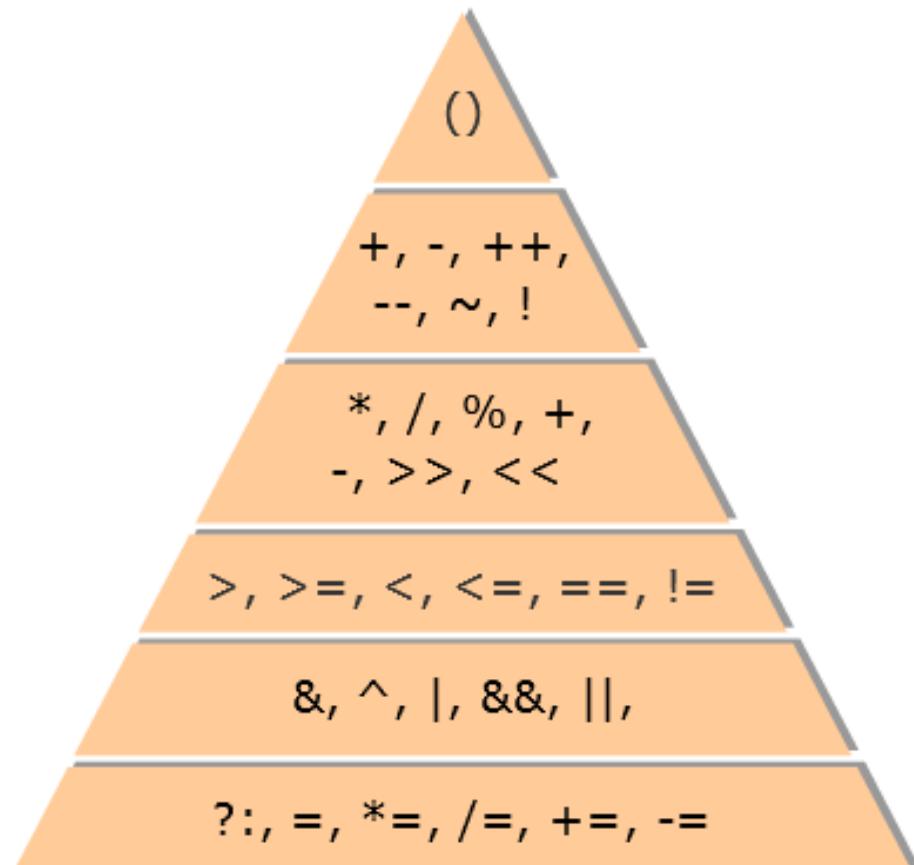
```
using System;
class Addition
{
    public static void Main()
    {
        byte numOne = 255;
        byte numTwo = 1;
        byte result = 0;
        try
        {
            unchecked
            {
                result = (byte)(numOne + numTwo);
            }
            Console.WriteLine("Result: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

- ◆ Expressions 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 Operators
  - ◆ Assignment Operators



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

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

## ◆ 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.

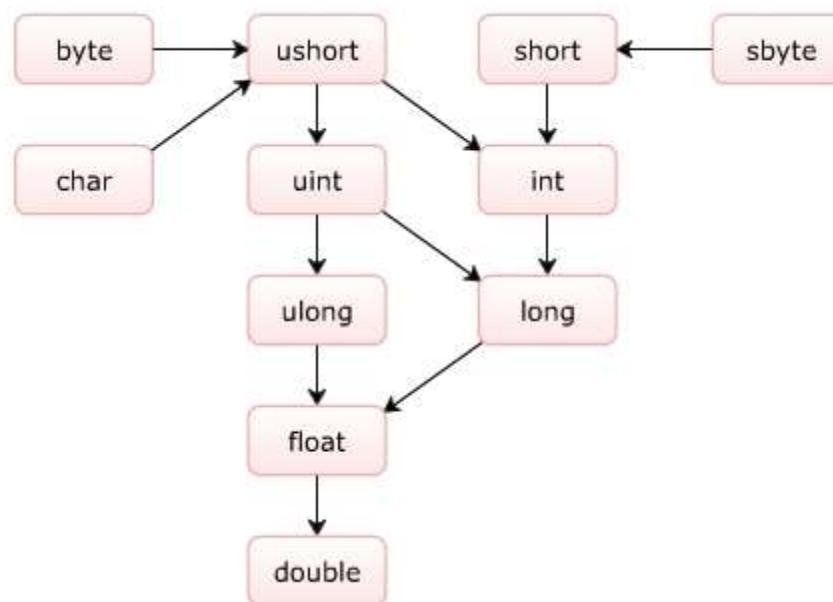
## Syntax

```
<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:

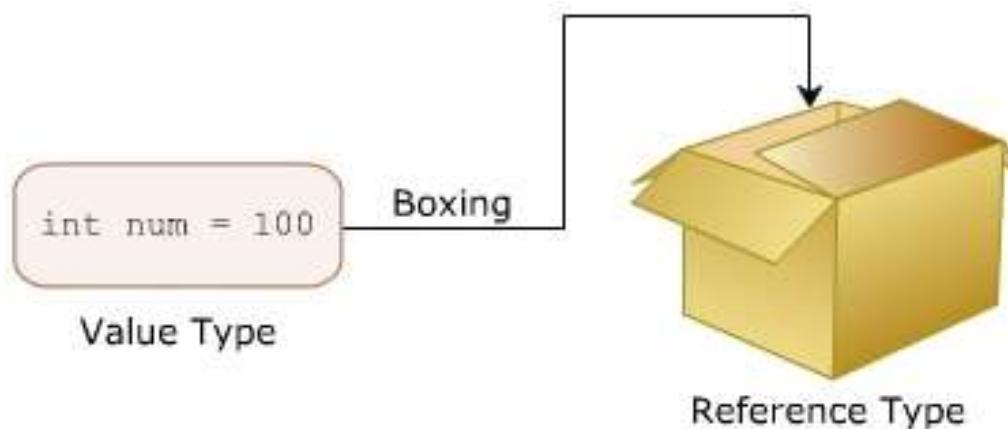
## Snippet

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

## Output

500.25

- ◆ Boxing 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:



- ◆ 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.

Session: 4

# C# Programming Constructs

- ◆ Explain selection constructs
- ◆ Describe loop constructs
- ◆ Explain jump statements in C#



- ◆ allows you to execute a block of statements after evaluating the specified logical condition.

## Syntax

```
if (condition)
{
    // one or more statements;
}
```

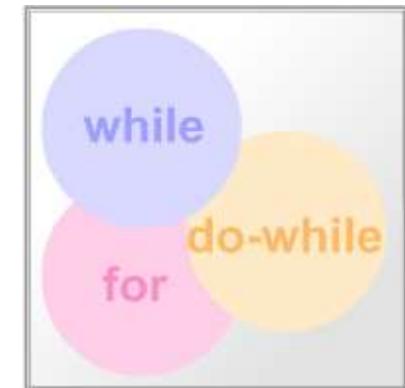
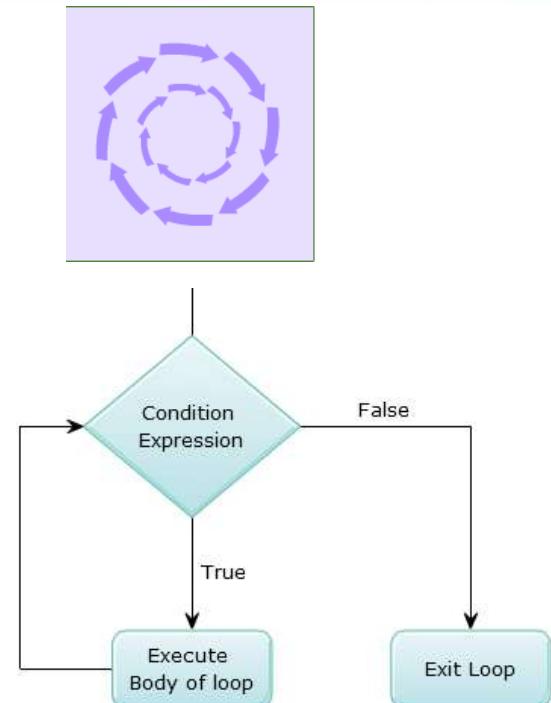
where,

- ◆ condition: Is the boolean expression.
- ◆ statements: Are set of executable instructions executed when the boolean expression returns true.

- ◆ A program is difficult to comprehend when there are too many **if** statements representing multiple selection constructs.
- ◆ To avoid that, in certain cases, the **switch...case** can be used.
- ◆ The **switch...case** statement is used when a variable needs to be compared against different values.

```
int day = 5;  
switch (day) {  
    case 1: Console.WriteLine("Sunday"); break;  
    case 2: Console.WriteLine("Monday"); break;  
    case 3: Console.WriteLine("Tuesday"); break;  
    default:  
        Console.WriteLine("Enter a number between 1 to 7");  
break;  
}
```

- ◆ allow to execute a single statement or a block of statements repetitively.
- ◆ contain a condition that identifies the number of times a specific block will be executed.
- ◆ If the condition is not specified, the loop continues infinitely : infinite loop.
- ◆ The loop constructs are also referred to as iteration statements.
- ◆ C# supports four types of loop constructs:
  - ❖ The **while** loop
  - ❖ The **do..while** loop
  - ❖ The **for** loop
  - ❖ The **foreach** loop



- ◆ The **while** loop is used to execute a block of code repetitively as long as the condition of the loop remains true.

## Syntax

```
while (condition)
{
    // one or more statements;
}
```

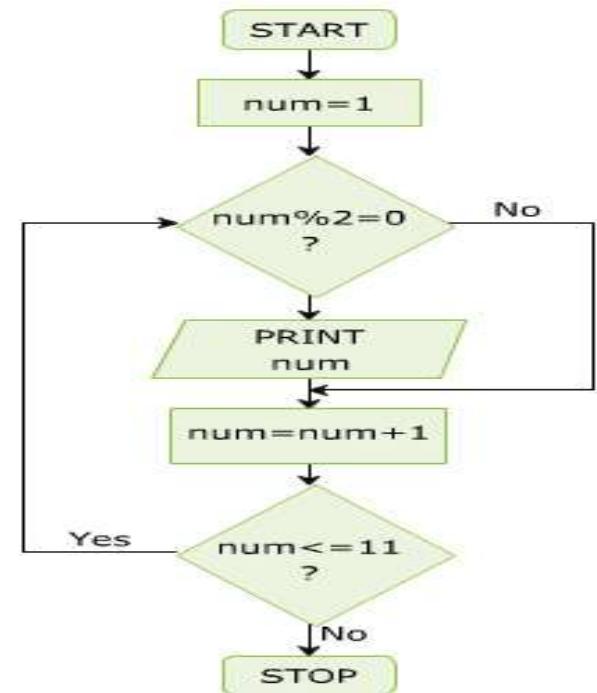
where:

- ◆ condition: Specifies the boolean expression.

- ◆ The **do-while** loop is similar to the **while** loop; however, it is always executed at least once without the condition being checked.

## Syntax

```
do
{
    // one or more statements;
} while (condition);
```



- ◆ is similar to the **while** statement in its function.
- ◆ The statements within the body of the loop are executed as long as the condition is true.

## Syntax

```
for (init; condition; increment/decrement)
{
    // one or more statements;
}
```

```
int num;

Console.WriteLine("Even Numbers");

for (num = 1; num <= 11; num++) {
    if ((num % 2) == 0) {
        Console.WriteLine(num);
    }
}
```

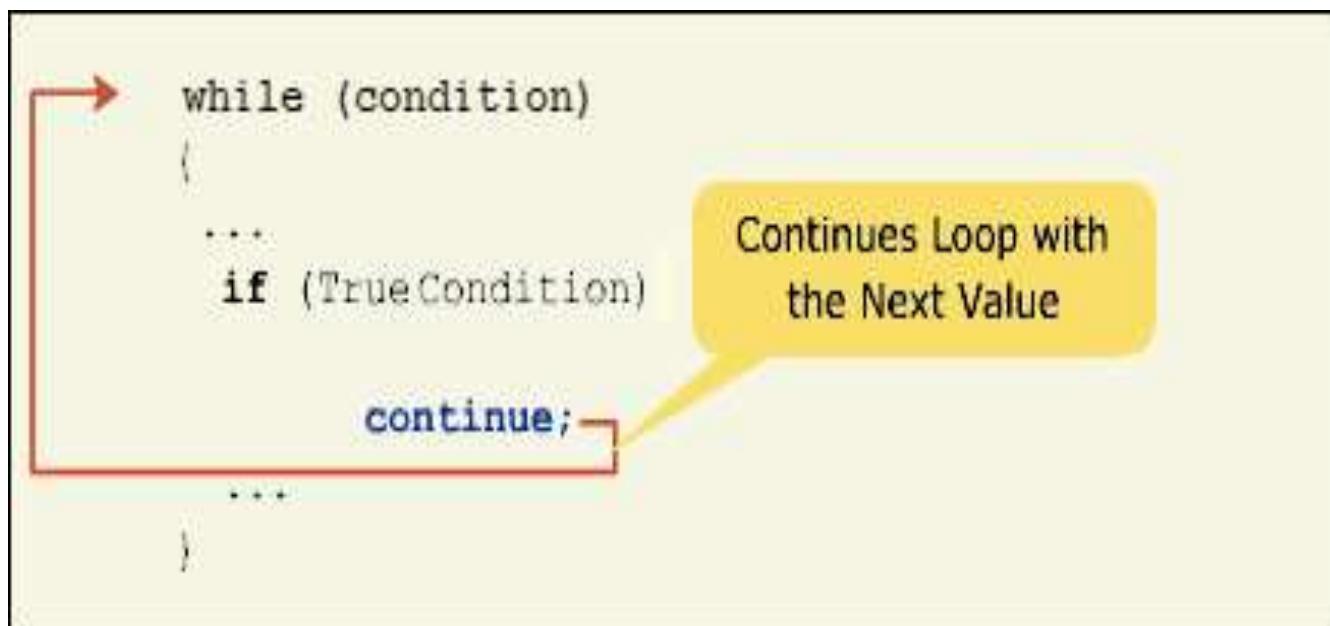
- ◆ is used in the selection and loop constructs.
- ◆ is most widely used in the **switch...case**, and in the **for** and **while** loops.
- ◆ In loops, it is used to exit the loop without testing the loop condition.

```
while (condition)
{
    ...
    if (TrueCondition)
        break;
    ...
}
```

Quit Loop

- ◆ is most widely used in the loop constructs
- ◆ is used to end the current iteration and transfer the program control back to the beginning of the loop.

The statements of the loop following the **continue** statement are ignored in the current iteration.



# The goto Statement

- ◆ allows to directly execute a labeled statement or block of statements.
- ◆ A labeled block or statement starts with a label.
- A label is an identifier ending with a colon.
- ◆ A single labeled block can be referred by more than one **goto** statements.

```
if (Truecondition)
{
    goto Display;
}
Display:←
Console.WriteLine("goto statement is executed");
```

Control Transferred to Display

## Snippet

```
int i = 0;
display:
    Console.WriteLine("Hello World");
    i++;
    if (i < 5) {
        goto display;
    }
```

- ◆ used to return a value of an expression or is used to transfer the control to the invoking method.
- ◆ must be the last statement in the method block.

The diagram illustrates the execution flow of a C# program. A red box highlights the `return;` statement within an `if` block. An arrow points from this `return;` statement to a yellow speech bubble containing the text "Program Terminates". Another red arrow points from the `Console.WriteLine("return statement")` line back towards the `return;` statement, indicating that the program exits before reaching that line.

```
if (TrueCondition)
{
    ...
    return;
}
else
{
    ...
    ...
}
Console.WriteLine("return statement")
```

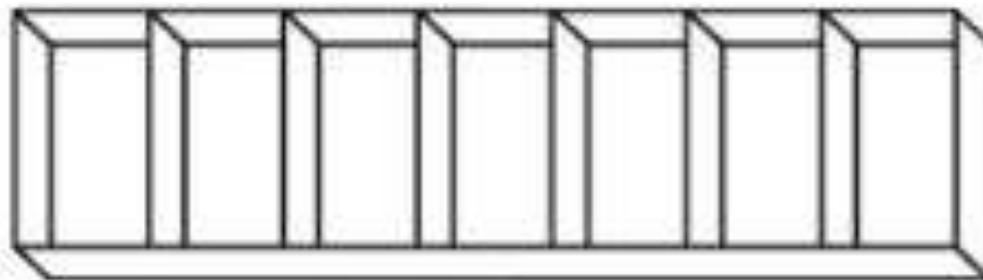
- ◆ Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- ◆ C# supports if...else, if...else...if, nested if, and switch...case selection constructs.
- ◆ Loop constructs execute a block of statement repeatedly for a particular condition.
- ◆ C# supports while, do-while, for, and foreach loop constructs.
- ◆ The loop control variables are often created for loops such as the for loop.
- ◆ Jump statements transfer the control to any labeled statement or block within a program.
- ◆ C# supports break, continue, goto, and return jump statements.

Session: 5

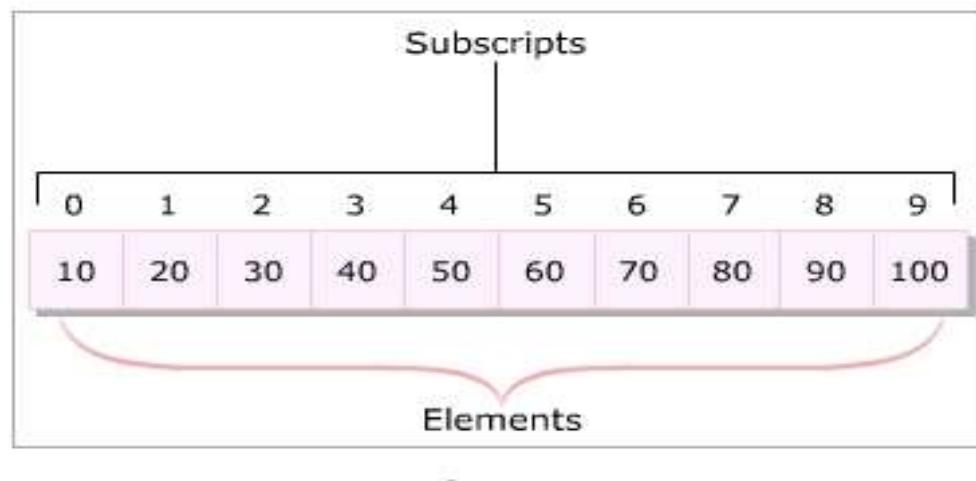
# Arrays

- ◆ Define and describe arrays
- ◆ List and explain the types of arrays
- ◆ Explain the Array class

- ◆ An array is a collection of elements of a single data type stored in adjacent memory locations.



## Example



- ◆ Arrays are reference type variables whose creation involves two steps:
  - ◆ **Declaration:** specifies the type of data that it can hold and an identifier.
  - ◆ **Memory allocation**
- ◆ Following is the syntax for declaring an array:

```
type [ ] arrayName ;
```

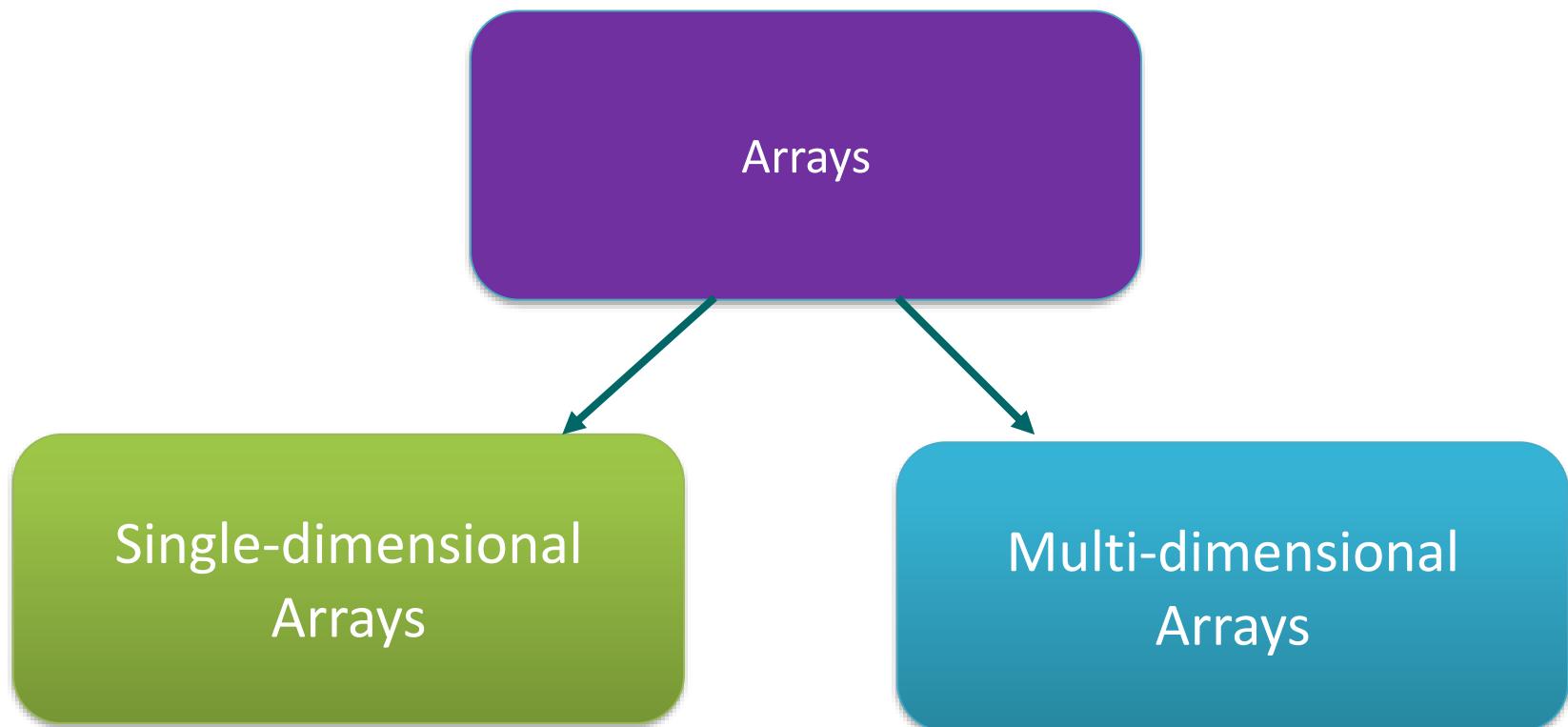
- ◆ The following syntaxes are used to initialise an array:

```
type[] arrayName;  
arrayName = new type[size-value];
```

```
type[] arrayName = new type[size-value];
```

```
type[] arrayIdentifier = {val1, val2, ..., valN};
```

- ◆ Based on how arrays store elements, arrays can be categorized into following two types:



- ◆ Elements are stored in a single row in allocated memory.
- ◆ Elements indexed from 0 to (n-1), where n is the total number of elements in the array.

## Example



- ◆ Syntax for declaring and initializing a single-dimensional array:

```
type[] arrayName;           //declaration  
arrayName = new type[length]; // creation
```

- ◆ Store combination of values of a single type in two or more rows and columns.
- ◆ Following are the two types of multi-dimensional arrays:

## Rectangular Array

- All the specified dimensions have constant values.
- Will always have the same number of columns for each row.

## Jagged Array

- One of the specified dimensions can have varying sizes.
- Can have unequal number of columns for each row.

```
type[,] <arrayName>; //declare rectangular array  
arrayName = new type[value1 , value2]; //initialize
```

- The following code demonstrates the use of jagged arrays

```
string[][] comp = new string[3][];

comp[0] = new string[] {"Intel", "AMD"};
comp[1] = new string[] {"IBM", "Microsoft", "Sun"};
comp[2] = new string[] {"HP", "Canon", "Lexmark", "Epson"};

for (int i=0; i<comp.GetLength(0); i++)
{
    Console.WriteLine("List of comp in gr" + (i+1) + ":\t");
    for (int j=0; j<comp[i].GetLength(0); j++)
    {
        Console.Write(comp[i][j] + " ");
    }
    Console.WriteLine();
}
```

# Fixed and Dynamic Arrays

## Fixed-length arrays

The number of elements is defined at the time of declaration.

For example, an array declared for storing days of the week will have exactly seven elements.

The number of elements is known and hence, can be defined at the time of declaration.

## Dynamic arrays

The size of the array can dynamically increase at runtime

For example, an array declared to store the e-mail addresses cannot have a predefined length.

Can add more elements to the array as and when required.  
Created using built-in classes of the .NET Framework.

- ◆ The **foreach** loop:
  - ◆ is an extension of the **for** loop.
  - ◆ used to perform specific actions on large data collections and can even be used on arrays.
  - ◆ Reads every element in the specified array.
  - ◆ Allows to execute a block of code for each element in the array.

## Syntax

```
foreach(type <identifier> in <list>)  
{  
    // statements  
}
```

- ◆ where:
  - ◆ **type**: Is the variable type.
  - ◆ **identifier**: Is the variable name.
  - ◆ **list**: Is the array variable name.

- ◆ Is a built-in class in the **System** namespace and is the base class for all arrays in C#.
- ◆ Provides methods for creating, searching, copying, and sorting arrays.

```
using System;  
  
class Subjects  
{  
    static void Main(string [] args)  
    {  
        Array objArray = Array.CreateInstance(typeof (string), 5);  
        objArray.SetValue("Marketing", 0);  
        objArray.SetValue("Finance", 1);  
        objArray.SetValue("Human Resources", 2);  
        objArray.SetValue("Information Technology", 3);  
        objArray.SetValue("Business Administration", 4);  
        for (int i = 0; i<= objArray.GetUpperBound(0); i++)  
        {  
            Console.WriteLine(objArray.GetValue(i));  
        }  
    }  
}
```

### Example

- ◆ Arrays are a collection of values of the same data type.
- ◆ C# supports zero-based index feature.
- ◆ There are two types of arrays in C#: Single-dimensional and Multi-dimensional arrays.
- ◆ A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- ◆ Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- ◆ The `Array` class defined in the `System` namespace enables to create arrays easily.
- ◆ The `Array` class contains the `CreateInstance()` method, which allows you to create single and multi-dimensional arrays.

Session: 6

# Classes and Methods

- ◆ Explain classes and objects
- ◆ Define and describe methods
- ◆ List the access modifiers
- ◆ Explain method overloading
- ◆ Define and describe constructors and destructors

- ◆ Programming languages are based on two fundamental concepts: data and ways to manipulate data.
- ◆ Traditional languages such as Pascal and C used the procedural approach which focused more on ways to manipulate data rather than on the data itself.
- ◆ This approach has several drawbacks such as lack of re-use and lack of maintainability.
- ◆ To overcome these difficulties, OOP was introduced, which focused on data.
- ◆ The object-oriented approach defines objects as entities having a defined set of values and a defined set of operations that can be performed on these values.

## Abstraction

- **Extracting only the required information from objects.**

For example, consider a television, it has a manual stating how to use the tv. However, this manual does not show all the technical details of the television.

## Encapsulation

- **Visible only some specific information of a class.**

Encapsulation also called data hiding.

Both abstraction and encapsulation are complementary to each other.

## Inheritance

- **Creating a new class based on an existing class.**

The existing class: base class.

The new class: derived class.

This is a very important concept of OOP as it helps to reuse the inherited attributes and methods.

## Polymorphism

- **The ability to behave differently in different situations.**

It is seen in programs where you have multiple methods declared with the same name but with different parameters and different behavior.



Identity: AXA 43 S

State: Color-Red, Wheels-Four

Behavior: Running

Identity: T002

State: Color-Brown

Behavior: Stable

- An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

- ◆ Several objects have a common state and behavior and thus, can be grouped under a single class.

## Example

- ◆ A Ford Mustang, a Volkswagen Beetle, and a Toyota Camry can be grouped together under the class Car. Here, Car is the class whereas Ford, Volkswagen, and Toyota are objects of the class Car.
- ◆ The following figure displays the class Car:

Car Class	
Characteristics	
➤ Make	
➤ Model	

Behavior	
➤ Driving	
➤ Accelerating	
➤ Braking	

```
class Students
{
    string _studName = "James Anderson";
    int _studAge = 27;

    void Display()
    {
        Console.WriteLine("Student Name: " + _studName);
        Console.WriteLine("Student Age: " + _studAge);
    }

    static void Main(string[] args)
    {
        Students objStudents = new Students();
        objStudents.Display();
    }
}
```

Fields

Methods

The diagram illustrates the structure of the `Students` class. It shows two highlighted fields: `_studName` and `_studAge`. It also highlights two methods: `Display()` and `Main(string[] args)`. A bracket labeled "Fields" points to the highlighted fields, and a bracket labeled "Methods" points to the highlighted methods.

- ◆ To access the variables and methods defined within class, using objects.
- ◆ An object is instantiated using the **new** keyword.

On encountering the new keyword, JIT compiler allocates memory for the object and returns a reference of that allocated memory.

## Syntax

```
<ClassName> <objectName> = new <ClassName>();
```

- ◆ Conventions to be followed for naming methods :

Cannot be a C# keyword, cannot contain spaces, and cannot begin with a digit

Can begin with a letter, underscore, or the "@" character

## Example

Invalid method names include **5Add**, **AddSum()**, **@int()**.

## Syntax

```
<access_modifier> <return_type> <MethodName> ([parameters])  
{  
    // body of the method  
}
```

## Parameters

- The variables included in a method definition are called parameters.

## Argument

- When the method is called, the data send into the method's parameters are called arguments.

### ◆ Example

```
class Student{  
    public void Display(string myParam)  
    {  
        ...  
        ...  
    }  
    public static void Main()  
    {  
        string myArg1 = "this is my argument";  
        ...  
        objStudent.Display(myArg1);  
    }  
}
```

Parameter

Argument

- The following code shows how to use optional arguments:

## Snippet

```
using System;
class OptParameterExample {
    void printMessage(String message = "Hello user!") {
        Console.WriteLine("{0}", message);
    }
    static void Main(string[] args) {
        OptParameterExample o = new OptParameterExample();
        o.printMessage("Welcome User!");
        o.printMessage();
    }
}
```

- ◆ cannot be inherited, consists of static data members and static methods only.
- The **static** keyword is used before the class name
- ◆ cannot create an instance of a static class using the **new** keyword. However, static constructors can be defined to initialize the static members.
  - ◆ Since there is no need to create objects of the static class to call the required methods, the implementation is simpler and faster.

- ◆ declared by using the **static** keyword.  
For example, the **Main ()** method is a static method and it does not require any instance of the class for it to be invoked.
- ◆ can directly refer only to static variables and methods of the class.
- ◆ also can refer to non-static methods and variables by using an instance of the class.

## Syntax

```
Static <return_type> <MethodName>()  
{  
    // body of the method  
}
```

- ◆ is a special variable, accessed without using an object of class.
- ◆ declared by the **static** keyword. When a static variable is created, it is automatically initialized before it is accessed.
- ◆ Only one copy of a static variable is shared by all the objects of the class.
- ◆ Therefore, a change in the value of such a variable is reflected by all the objects of the class.

```
class Employee
{
    public static int EmpId = 20 ;
    public static string EmpName = "James";

    static void Main (string[] args)
    {

        Console.WriteLine ("Employee ID: " + EmpId);
        Console.WriteLine ("Employee Name: " + EmpName);
    }
}
```

- ◆ In C#, there are four commonly used access modifiers.



- ◆ **public:**  
provides the most permissive access level.
- ◆ **private:**  
provides the least permissive access level. Private members are accessible only within the class in which they are declared.
- ◆ **protected:**  
Protected the class members are accessible within the class as well as within the derived classes.
- ◆ **internal:**  
Internal members are accessible only within the classes of the same assembly.

- ◆ Causes arguments to be passed to called method by reference.
- ◆ In **call by reference**, the called method changes the value of the arguments passed to it.
- ◆ Both the called method and the calling method must explicitly specify the **ref, out** keyword before the required parameters.
- ◆ **out** keyword does **not require** the arguments to be initialized.

```
classRefParameters
{
    static void Calculate(ref int numValueOne, ref int
    numValueTwo)
    {
        numValueOne = numValueOne * 2;
        numValueTwo = numValueTwo / 2;
    }
}
```

```
classOutParameters
{
    static void Depreciation(out int val)
    {
        val = 20000;
        int dep = val * 5/100;
        int amt = val - dep;
        Console.WriteLine("Depreciation Amount: " + dep);
        Console.WriteLine("Reduced value after depreciation: " +
        amt);
    }
}
```

- ◆ In OOP, every method has a signature which includes:

- The number of parameters passed to the method, the data types of parameters and the order in which the parameters are written.
- the signature of the method is written in parentheses next to the method name.
- No class is allowed to contain two methods with the same name and same signature, but it is possible for a class to have two methods having the same name but different signatures.
- The concept of declaring more than one method with the same method name but different signatures is called method overloading.

## Method Overloading in C# 2-2

- The following figure displays the concept of method overloading using an example:

```
int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo)
{
    int result = valOne + valTwo;
    return result;
}
```



Not Allowed in C#

```
int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo, int valThree)
{
    return valOne + valTwo + valThree;
}
```



Allowed in C#

- ◆ The **this** keyword is used to refer to the current object of the class to resolve conflicts between variables having same names and to pass the current object as a parameter.
- ◆ You cannot use the **this** keyword with static variables and methods.

```
class Dimension
{
    double _length;
    double _breadth;
    public double Area(double _length, double _breadth)
    {
        this._length = _length;
        this._breadth = _breadth;
        return _length * _breadth;
    }
}
```

- ◆ A C# class can define constructors and destructors as follows:

## Constructors

A C# class can contain one or more special member functions having the same name as the class, called constructors.

A constructor is a method having the same name as that of the class.

## Destructors

A C# class can also have a destructor (only one is allowed per class), which is a special method and also has the same name as the class but prefixed with a special symbol ~.

A destructor of an object is executed when the object is no longer required in order to de-allocate memory of the object.

# Default and Static Constructors

## Default Constructors

C# creates a default constructor for a class if no constructor is specified within the class.

The default constructor automatically initializes all the numeric data type instance variables to zero.

If you define a constructor in the class, the default constructor is no longer used.

## Static Constructors

A static constructor is used to initialize static variables and to perform a particular action only once.

It is invoked before any static member of the class is accessed.

A static constructor does not take any parameters and does not use any access modifiers because it is invoked directly by the CLR instead of the object.

- The following code demonstrates the use of constructor overloading:

```
using System;
public class Rectangle {
    double _length;
    double _breadth;
    public Rectangle() {
        _length = 13.5;
        _breadth = 20.5;
    }
    public Rectangle(double len, double wide) {
        _length = len;
        _breadth = wide;
    }

    public double Area() {
        return _length * _breadth;
    }
    static void Main(string[] args) {
        Rectangle objRect1 = new Rectangle();
        Console.WriteLine("Area of rectangle = " + objRect1.Area());
        Rectangle objRect2 = new Rectangle(2.5, 6.9);
        Console.WriteLine("Area of rectangle = " + objRect2.Area());
    }
}
```

- ◆ The programming model that uses objects to design a software application is termed as OOP.
- ◆ A method is defined as the actual implementation of an action on an object and can be declared by specifying the return type, the name and the parameters passed to the method.
- ◆ It is possible to call a method without creating instances by declaring the method as static.
- ◆ Access modifiers determine the scope of access for classes and their members.
- ◆ The four types of access modifiers in C# are public, private, protected and internal.
- ◆ Methods with same name but different signatures are referred to as overloaded methods.
- ◆ In C#, a constructor is typically used to initialize the variables of a class.

Session: 7

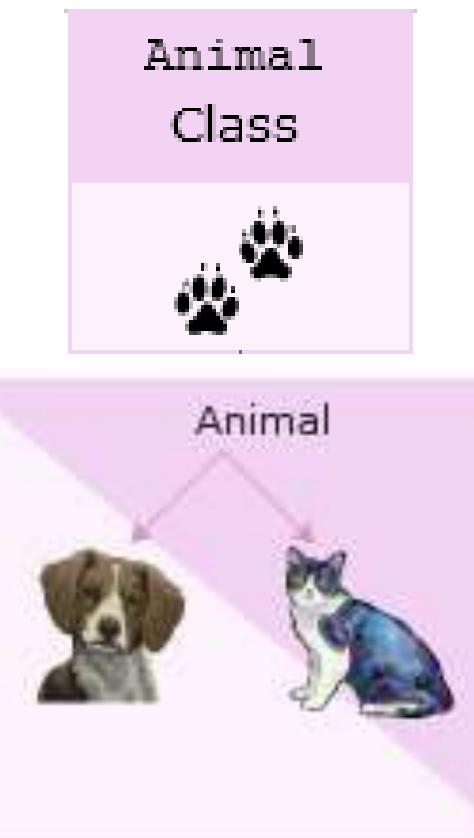
# Inheritance and Polymorphism

- ◆ Define and describe inheritance
- ◆ Explain method overriding
- ◆ Define and describe sealed classes
- ◆ Explain polymorphism

- ◆ The purpose of inheritance is to reuse common methods and attributes among classes without recreating them.
- ◆ Reusability enables to use the same code in different applications with little or no changes.

Example:

- ◆ Consider a class named **Animal** which defines attributes and behavior for animals.
- ◆ If a new class named **Cat** has to be created, it can be done based on **Animal**.



- ◆ To derive a class from another class, insert a colon after the name of the derived class, followed by the name of the base class.
- ◆ The derived class can now inherit all non-private methods and attributes of the base class.
- ◆ The following syntax is used to inherit a class in C#:

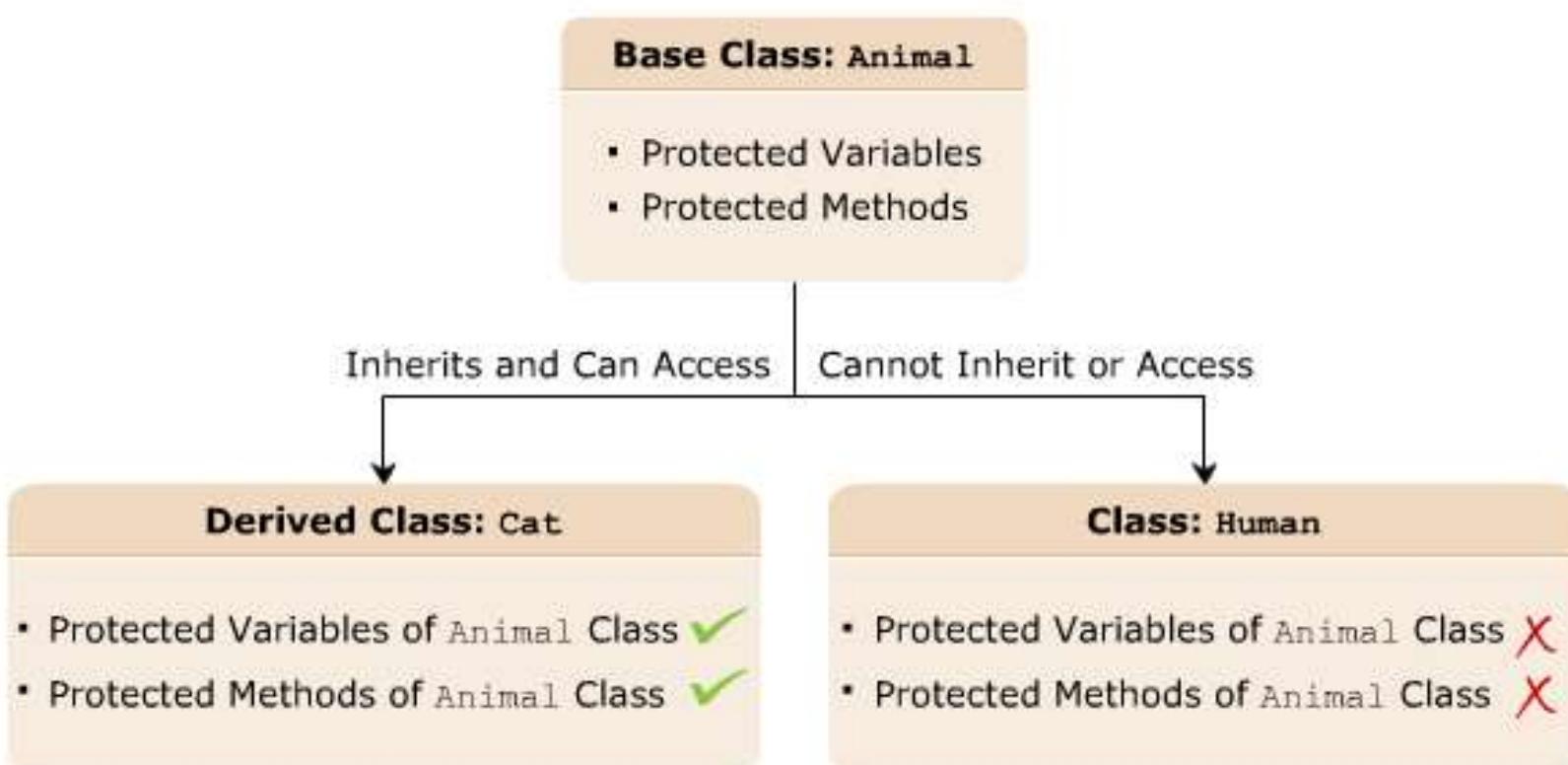
```
<DerivedClassName> : <BaseClassName>
```

where,

- ◆ **DerivedClassName**: Is the name of the newly created child class.
- ◆ **BaseClassName**: Is the name of the parent class

# protected Access Modifier

- ◆ used to protect the members in a class so that they are accessed only by the class they declared or by the child class.



- The following syntax shows the use of the **base** keyword:

```
class <Parent>{
    <accessmodifier> <return type> <MethodA>() {...}
}

class <Child>:<Parent> {
    <accessmodifier> <return type> <MethodA>() {
        ...
        base.<MethodA>();
    }
}
```

```
class Animal
{
    public void Eat(){}
}
class Dog : Animal
{
    public void Eat(){}
    public static Main(string args[])
    {
        Dog objDog = new Dog();
        objDog.Eat();
        base.Eat();
    }
}
```

- ◆ Used as a modifier in order to hide the base class method :

```
class <Parent> {  
    <access modifier> <returntype> <BaseMethod>() {}  
}  
  
class <Child>:<Parent> {  
    new <access modifier> <returntype> <BaseMethod>() {}  
}
```

- ◆ Used as an operator to creates an object :

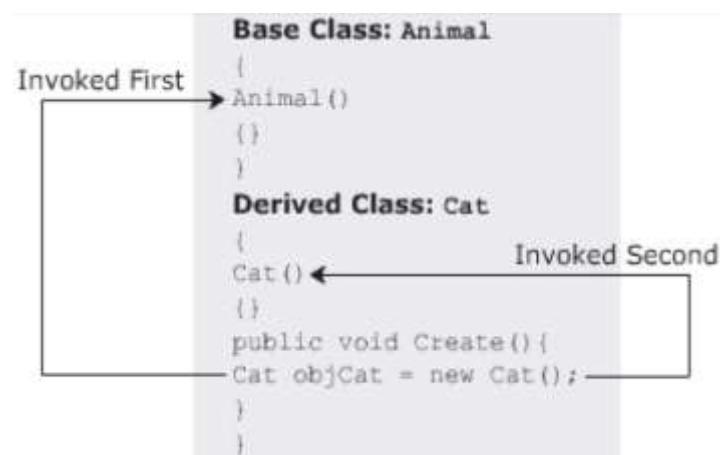
```
Employees objEmp = new Employees();
```

- ◆ In C#:

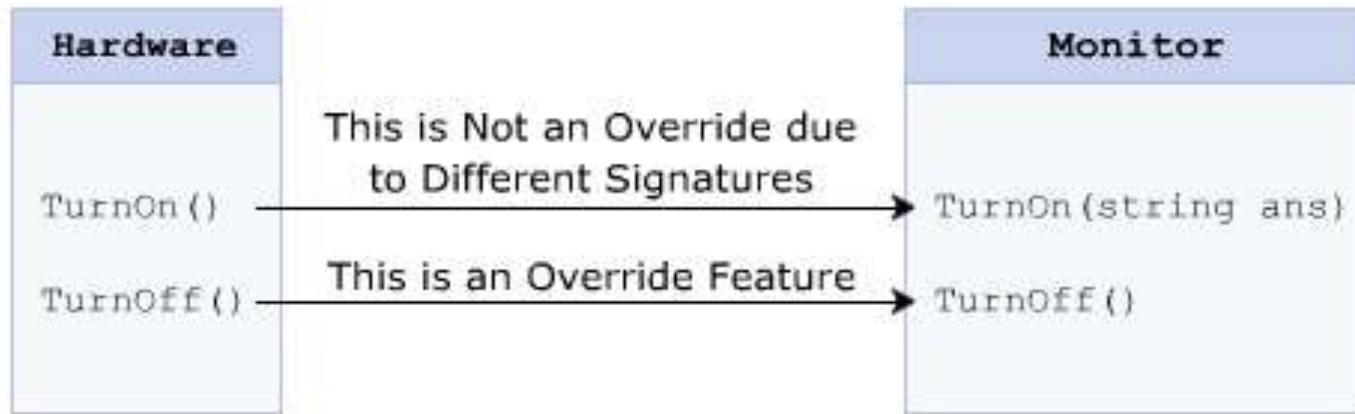
the base class constructor can be invoked by either instantiating the derived class or the base class  
the constructor of base class goes first, and then constructor of derived class.

the base class constructor can be invoked by using base keyword in the derived class constructor declaration.

- ◆ However, constructors cannot be inherited



- ◆ Allows a method in derived class having the same name and signature as the method in the base class, to be rewritten to redefine new behavior.
- ◆ Ensures reusability while inheriting classes.
- ◆ Is implemented in the derived class , is known as the Overridden Base Method.
- ◆ The following figure depicts method overriding:



# virtual and override Keywords

- ◆ To implement the **overriding** feature, using the **virtual** and **override** keywords:

```
<access_modifier> virtual <return_type> <MethodName>(<parameters>);
```

```
<access_modifier> override <return_type> <MethodName>(<parameters>);
```

where,

- ◆ **virtual**: Is a keyword used to declare a method in the base class that can be overridden by the derived class.
- ◆ **override**: Is a keyword used to declare a method in the derived class

- ◆ A sealed class is a class that prevents inheritance.
- ◆ The features :

be declared by preceding the class keyword with the **sealed** keyword.

cannot be a base class

## Syntax

```
sealed class <ClassName>
{
    //body of the class
}
```

prevents the method from further overriding by derived class.

When the derived class overrides a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed.

- ◆ The following syntax is used to declare an overridden method as sealed:

```
sealed override <return_type> <MethodName>() {  
}
```

- ◆ Polymorphism is the ability of an entity to behave differently in different situations.
- ◆ Polymorphism is derived from two Greek words, namely **Poly** and **Morphos**, meaning forms. Polymorphism means existing in multiple forms.
- ◆ Polymorphism allows methods to function differently based on the parameters and their data types. For example:

The diagram illustrates polymorphism through method overloading. It shows a class named **Shape** with two methods, both named **SetDimensions**. The first method takes **int length** and **int breadth** as parameters. The second method takes **int length**, **int breadth**, and **int height** as parameters. A large curly brace to the right of the methods is labeled "Overloaded Methods – Way of Implementing Polymorphism".

```
Shape  
SetDimensions(int length,  
              int breadth)  
SetDimensions(int length,  
              int breadth,  
              int height)}
```

Overloaded Methods –  
Way of Implementing  
Polymorphism

- ◆ Inheritance allows to create a new class from another class, thereby inheriting its common properties and methods.
- ◆ Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.
- ◆ Method overriding is a process of redefining the base class methods in the derived class.
- ◆ Methods can be overridden by using a combination of virtual and override keywords within the base and derived classes respectively.
- ◆ Sealed classes are classes that cannot be inherited by other classes, declared by using the sealed keyword.
- ◆ Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.

Session: 8

# Abstract Classes and Interfaces

- ◆ Define and describe abstract classes
- ◆ Explain interfaces
- ◆ Compare abstract classes and interfaces

- ◆ Is a class specifically to be used as a incomplete base class.
- ◆ cannot be instantiated, but be implemented or derived.
- ◆ may contain one or more of the following:
  - ◆ normal data member(s) / method(s)
  - ◆ abstract method(s)
- ◆ Declared by keyword **abstract**
- ◆ Syntax:

```
public abstract class <ClassName>
{
    <accModifier> abstract <returnType> <MethodName>(pars) ;
}
```

- The following code declares and implements an abstract class:

```
abstract class Animal {  
    public void Eat() {  
        Console.WriteLine("eats food in order to survive");  
    }  
    public abstract void AnimalSound();  
}  
  
class Lion : Animal {  
    public override void AnimalSound() {  
        Console.WriteLine("Lion roars");  
    }  
}
```

- ◆ contain only abstract members
- ◆ cannot be instantiated but can be inherited by classes or other interfaces.
- ◆ declared by the keyword **interface**.
- ◆ In C#, by default, all members declared in an interface have **public** access modifier.
- ◆ The following figure displays an example of an interface:

Animal Abstract Class	IAnimalInterface
<pre>Eat () {     "Every animal eats food"; } Habitat (); AnimalSound ();</pre>	<pre>Eat (); //No Body Habitat (); AnimalSound ();</pre>

```
interface Ianimal { void Habitat(); }

class Dog : Ianimal {
    public void Habitat() {
        Console.WriteLine("Can be housed with human
        beings");
    }

    static void Main(string[] args) {
        Dog objDog = new Dog();
        Console.WriteLine(objDog.GetType().Name);
        objDog.Habitat();
    }
}
```

## Output

Dog

Can be housed with human beings

# Interfaces and Multiple Inheritance

```
interface ITerrestrialAnimal { void Eat(); }
interface IMarineAnimal { void Swim(); }
class Crocodile : ITerrestrialAnimal, IMarineAnimal {
    public void Eat() {
        Console.WriteLine("The Crocodile eats flesh");
    }
    public void Swim() {
        Console.WriteLine("The Crocodile can swim 4 times faster
                          than an Olympic swimmer");
    }
    static void Main(string[] args) {
        Crocodile o = new Crocodile();
        o.Eat();
        o.Swim();
    }
}
```

## Output

The Crocodile eats flesh  
The Crocodile can swim four times faster than an  
Olympic swimmer

- ◆ The following syntax is used to inherit an interface:

## Syntax

```
interface <InterfaceName> : <Inherited_InterfaceName>
{
    // method declaration;
}
```

- ◆ where,
  - ◆ **InterfaceName**: name of the interface that inherits another interface.
  - ◆ **Inherited\_InterfaceName**: name of the inherited interface.

```
interface IAnimal { void Drink(); }
interface ICarnivorous { void Eat(); }
interface IReptile:IAnimal, ICarnivorous { void Habitat(); }
class Crocodile : IReptile {
    public void Drink() {
        Console.WriteLine("Drinks fresh water");
    }
    public void Habitat() {
        Console.WriteLine("Can stay in Water and Land");
    }
    public void Eat() {
        Console.WriteLine("Eats Flesh");
    }
    static void Main(string[] args) {
        Crocodile o = new Crocodile();
        Console.WriteLine(o.GetType().Name);
        o.Habitat();
        o.Eat();
        o.Drink();
    }
}
```

Snippet

Output

```
Crocodile
Can stay in Water and Land
Eats Flesh
Drinks fresh water
```

# The **is** and **as** Operators in Interfaces 1-2

- ◆ The **is** and **as** operators when used with interfaces, verify whether the specified interface is implemented or not.

## is Operator

Checks the compatibility between two types returns a boolean value based on the check operation performed.

## as Operator

performs conversion between compatible types returns null if the two types are not compatible with each other.

## The is and as Operators in Interfaces 2-2

```
interface ICalculate { double Area(); }

class Rectangle : ICalculate{
    float _length, float _breadth;
    public Rectangle(float valOne, float valTwo) {
        _length = valOne;
        _breadth = valTwo;
    }
    public double Area() { return _length * _breadth; }
    static void Main(string[] args) {
        Rectangle objRectangle = new Rectangle(10.2F, 20.3F);
        if (objRectangle is ICalculate) {
            Console.WriteLine("Area : {0:F2}" ,
                objRectangle.Area());
        } else {
            Console.WriteLine("Interface method not implemented");
        }
    }
}
```

# Differences Between an Abstract Class and an Interface

- ◆ Abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class.
- ◆ However, there are certain differences between them as shown in the following table:

Abstract Classes	Interfaces
can inherit a class and multiple interfaces.	can inherit multiple interfaces but cannot inherit a class.
can have methods with a body.	cannot have methods with a body.
is implemented using the override keyword.	is implemented without using the override keyword.
is a better option when you need to implement common methods and declare common abstract methods.	is a better option when you need to declare only abstract methods.
can declare constructors and destructors.	cannot declare constructors or destructors.

# Recommendations for Using Abstract Classes and Interfaces

## Abstract class

- create reusable programs and maintain multiple versions of these programs
- helps to maintain the version of the programs in a simple manner.
- must exist a relationship between the abstract class and the classes that inherit the abstract class.

## Interface

- create different methods that are useful for different types of objects
- are suitable for implementing similar functionalities in dissimilar classes.
- cannot be changed once they are created.
- A new interface needs to be created to create a new version of the existing interface

- ◆ An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- ◆ IntelliSense provides access to member variables, functions, and methods of an object or a class.
- ◆ When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- ◆ A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- ◆ A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- ◆ Re-implementation occurs when the method declared in the interface is implemented in a class using the virtual keyword and this virtual method is then overridden in the derived class.
- ◆ The is operator is used to check the compatibility between two types or classes and as operator returns null if the two types or classes are not compatible with each other.

Session: 9

# Properties and Indexers

- ◆ Define properties in C#
- ◆ Explain properties, fields, and methods
- ◆ Explain indexers

# Applications of Properties

- ◆ Properties:

- ◆ allow to access private fields and ensure security of them
- ◆ can validate data before making changes to the protected fields and also perform specified actions on those changes.
- ◆ support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.

```
<access_modifier> <return_type> <PropertyName> {  
    get {  
        // return value  
    }  
    set {  
        // assign value  
    }  
}
```

where, **return\_type**: the type of data the property will return.

- ◆ allow to read and assign a value to a field:

## The get accessor

- used to read a value
- executed when the property name is referred.
- does not take any parameter and returns a value that is of the return type of the property.

## The set accessor

- used to assign a value
- executed when the property is assigned a new value.
- stored new value into a private field by an implicit parameter called **value** (keyword in C#)

- ◆ Properties are broadly divided into three categories:



- ◆ The static property is:
  - ◆ declared by using the **static** keyword.
  - ◆ accessed using the class name.
  - ◆ used to access and manipulate static fields of a class in a safe manner.
- ◆ The following code demonstrates a class with a static property.

```
class University {  
    private static string _department;  
    private static string _universityName;  
    public static string Department {  
        get {  
            return _department;  
        }  
        set {  
            _department = value;  
        }  
    }  
}
```

- ◆ Declared by using the **abstract** keyword.
- ◆ Contain only the declaration of the property without the body of the **get** and **set** accessors (can be implemented in the derived class).
- ◆ Are only allowed in an abstract class.
- ◆ Are used :
  - ❖ to secure data within multiple fields of the derived class of the abstract class.
  - ❖ to avoid redefining properties by reusing the existing properties.

```
public abstract class Figure {  
    public abstract float DimensionOne {  
        set;  
    }  
    public abstract float DimensionTwo {  
        set;  
    }  
}
```

# Auto-Implemented Properties

- ◆ Are properties **without** explicitly **providing** the **get** and **set** accessors.
- ◆ For an auto-implemented property, the compiler automatically creates:
  - ◆ a private field to store the property variable.
  - ◆ the corresponding **get** and **set** accessors.
- ◆ Syntax:
- ◆ Example.

```
public string Name { get; set; }
```

```
class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

# Object Initializers

- The following code uses object initializers to initialize an **Employee** object.

```
class Employee {  
    public string Name { get; set; }  
    public int Age { get; set; }  
    public string Designation { get; set; }  
    static void Main (string [] args) {  
        Employee emp1 = new Employee {  
            Name = "John Doe",  
            Age = 24,  
            Designation = "Sales Person"  
        };  
        Console.WriteLine("Name: {0}, Age: {1}, Designation:  
        {2}", emp1.Name, emp1.Age, emp1.Designation);  
    }  
}
```

## Output

- Name: John Doe, Age: 24, Designation: Sales Person

# Properties vs Fields

Properties	Fields
are data members that can assign and retrieve values.	are data members that store values.
cannot be classified as variables and therefore, cannot use the ref and out keywords.	are variables that can use the ref and out keywords.
are defined as a series of executable statements.	can be defined in a single statement.
are defined with two accessors or methods, the get and set accessors.	are not defined with accessors.
can perform custom actions on change of the field's value.	are not capable of performing any customized actions.

# Properties vs Methods

Properties	Methods
represent characteristics of an object.	represent the behavior of an object.
contain two methods which are automatically invoked without specifying their names.	are invoked by specifying method names along with the object of the class.
cannot have any parameters.	can include a list of parameters.
can be overridden but cannot be overloaded.	can be overridden as well as overloaded.

- ◆ allow instances of a class or struct to be indexed like arrays.
- ◆ are syntactically similar to properties, but unlike properties, the accessors of indexers accept one or more parameters.

## Example

- ◆ Consider a high school teacher who wants to go through the records of a particular student to check the student's progress.
- ◆ Calling the appropriate methods every time to set and get a particular record makes the task tedious.
- ◆ Creating an indexer for student ID:
  - ◆ makes the task of accessing the record much easier as indexers use index position of the student ID to locate the student record.

Indexer	Student_Details	
StudentID	StudentID	StudentName
S001	S003	John
S002	S002	Smith
S003	S004	Albert
S004	S001	Rosa



- ◆ An indexer can be defined by specifying the following:
  - ❖ An access modifier, which decides the scope of the indexer.
  - ❖ The return type of the indexer, which specifies the type of value an indexer will return.
  - ❖ The **this** keyword, which refers to the current instance of the current class.
  - ❖ The bracket notation ( [ ] ), which consists of the data type and the identifier of the index.
  - ❖ The open and close curly braces, which contain the declaration of the **set** and **get** accessors.

## Syntax

```
<access_modifier> <return_type> this [<parameter>]  
{  
    get { // return value }  
    set { // assign value }  
}
```

# Implementing Inheritance

- ◆ Indexers can be inherited like other members of the class.

Snippet

```
class Numbers {
    private int[] num = new int[3];
    public int this[int index] {
        get { return num [index]; }
        set { num [index] = value; }
    }
}
class EvenNumbers : Numbers {
    public static void Main() {
        EvenNumbers objEven = new EvenNumbers();
        objEven[0] = 0;
        objEven[1] = 2;
        objEven[2] = 4;
        for(int i=0; i<3; i++) {
            Console.WriteLine(objEven[i]);
        }
    }
}
```

# Indexers in Interfaces

```
public interface Idetails {
    string this[int index] { get; set; }
}

class Students : Idetails {
    string [] studentName = new string[3];
    public string this[int index] {
        get { return studentName[index]; }
        set { studentName[index] = value; }
    }
    static void Main(string[] args) {
        Students objStudent = new Students();
        objStudent[0] = "James";
        objStudent[1] = "Wilson";
        objStudent[2] = "Patrick";
        Console.WriteLine("Student Names");
        Console.WriteLine();
        for (int i = 0; i< 3; i++) {
            Console.WriteLine(objStudent[i]);
        }
    }
}
```

# Difference between Properties and Indexers

- ◆ Indexers are syntactically similar to properties.
- ◆ However, there are certain differences between them.

Properties	Indexers
are assigned a unique name	cannot be assigned a name and use the <b>this</b> keyword.
are invoked using the specified name.	are invoked through an index of the created instance.
can be declared as <b>static</b> .	can never be declared as <b>static</b> .
without parameters.	are declared with at least one parameter.
cannot be overloaded.	can be overloaded.
<b>Overridden properties</b> are accessed using the syntax <b>base</b> . <b>Prop</b> , where <b>Prop</b> is the name of the property.	are accessed using the syntax <b>base</b> [ <b>indExp</b> ], where <b>indExp</b> is the list of parameters separated by commas.

- ◆ Properties protect the fields of the class while accessing them.
- ◆ Property accessors enable you to read and assign values to fields.
- ◆ A field is a data member that stores some information.
- ◆ Properties enable you to access the private fields of the class.
- ◆ Methods are data members that define a behavior performed by an object.
- ◆ Indexers treat an object like an array, thereby providing faster access to data within the object.
- ◆ Indexers are syntactically similar to properties, except that they are defined using the this keyword along with the bracket notation ([]).

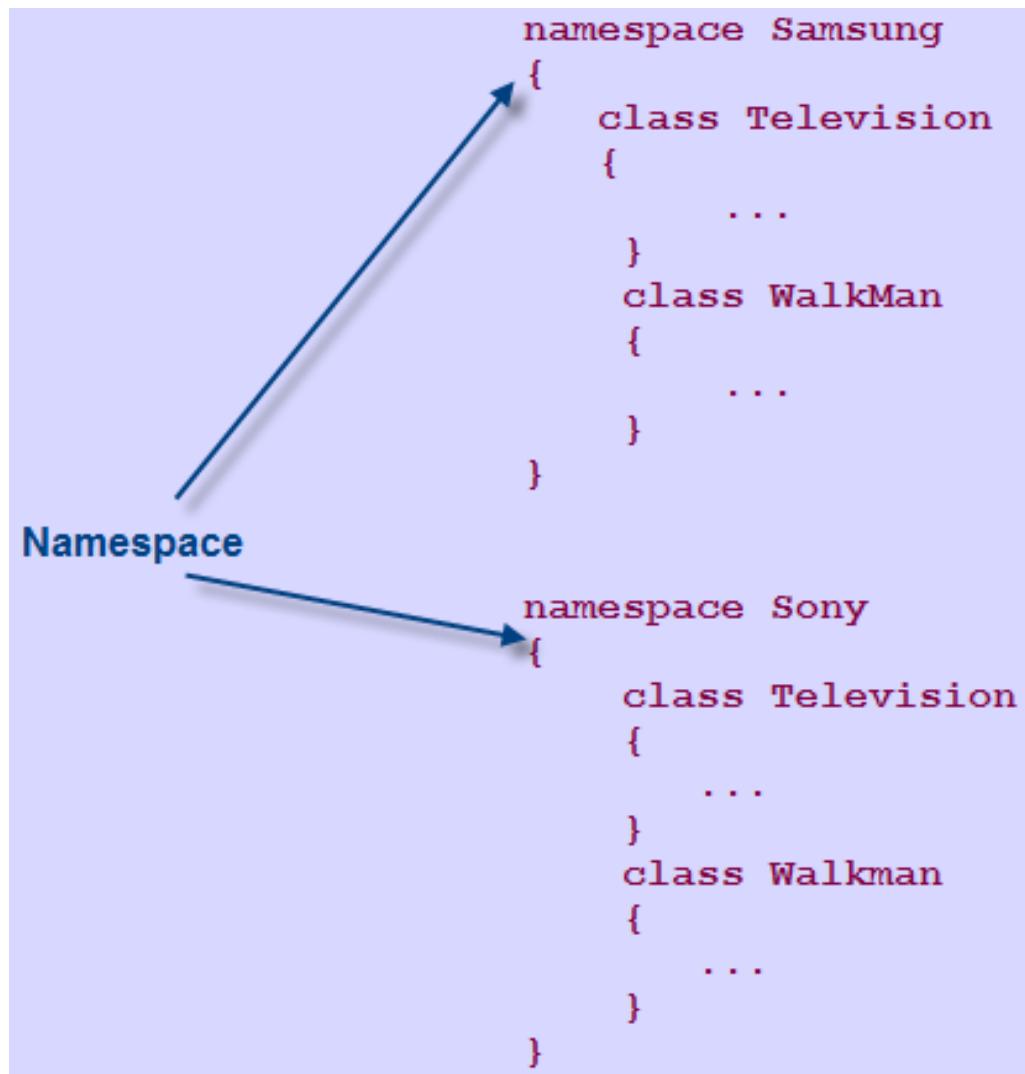
Session: 10

# Namespaces

- ◆ Define and describe namespaces
- ◆ Explain nested namespaces

- ◆ A namespace:

- ◆ Is used to group classes logically and prevent name clashes between classes with identical names.
- ◆ Reduces any complexities when the same program is required in another application.



# Using Namespaces

- ◆ allows to specify a unique identifier for each namespace.
- ◆ helps you to access the classes within the namespace.
- ◆ apart from classes, the following structures can be declared in a namespace:

Interface

- is a reference type that contains declarations of the events, indexers, methods, and properties.

Structure

- is a value type
- can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.

Enumeration

- is a value type that consists of a list of named constants.

Delegate

- is a reference type that refers to one or more methods.
- can be used to pass data as parameters to methods.

## System.Collections

- contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries.

## System.Data

- contains classes that make up the ADO.NET architecture.
- The ADO.NET architecture allows to build components that can be used to insert, modify, and delete data from multiple data sources.

## System.Diagnostics

- contains classes that are used to interact with the system processes.
- also provides classes that are used to debug applications and trace the execution of the code.

## System.IO

- contains classes that enable you to read from and write to data streams and files.

## System.Web

- contains classes that allow you to create Web-based applications.

## System.Net

- provides classes and interfaces that allow communication between the browser and the server.

# Using the System Namespace

- The two approaches of referencing the System namespace are:

Class 1

```
using System;
```

Class 2

```
System.Console.Read();  
...  
...  
...  
System.Console.Read();  
...  
System.Console.Write();
```

**Efficient Programming**

**Inefficient Programming**

- Though both are technically valid, the first approach is more recommended.

# Using the system-defined Namespace

- ◆ The following syntax is used to access a method in a system-defined namespace:

```
<NamespaceName>.<ClassName>.<MethodName>;
```

- ◆ In the syntax:
  - ❖ NamespaceName: Is the name of the namespace.
  - ❖ ClassName: Is the name of the class that you want to access.
  - ❖ MethodName: Is the name of the method within the class that is to be invoked.

## Snippet

```
using System;
namespace Automotive
{
    public class SpareParts {
        string _spareName;
        public SpareParts() {
            _spareName = "Gear Box";
        }
        public void Display() {
            Console.WriteLine("Spare Part name: _spareName");
        }
    }
}
```

```
using System;
using Students;
namespace Students {
    class StudentDetails {
        string _studName = "Alexander";
        int _studID = 30;
        public StudentDetails() {
            Console.WriteLine("Student Name: " + _studName);
            Console.WriteLine("Student ID: " + _studID);
        }
    }
}

namespace Examination {
    class ScoreReport {
        public string Subject = "Science";
        public int Marks = 60;
        static void Main(string[] args) {
            StudentDetails objStudents = new StudentDetails();
            ScoreReport objReport = new ScoreReport();
            Console.WriteLine("Subject: " + objReport.Subject);
            Console.WriteLine("Marks: " + objReport.Marks);
        }
    }
}
```

```
using System;
namespace Students {
    class StudentDetails {
        string _studName = "Alexander";
        int _studId = 30;
        public StudentDetails() {
            Console.WriteLine("Student Name: " + _studName);
            Console.WriteLine("Student ID: " + _studId);
        }
    }
}
namespace Examination {
    class ScoreReport {
        public string Subject = "Science";
        public int Marks = 60;
        static void Main(string[] args) {
            Students.StudentDetails os = new Students.StudentDetails();
            ScoreReport or = new ScoreReport();
            Console.WriteLine("Subject: " + or.Subject);
            Console.WriteLine("Marks: " + or.Marks);
        }
    }
}
```

# Implementing Nested Namespaces

- ◆ C# allows to create a hierarchy of namespaces by creating namespaces within namespaces.
- ◆ Such nesting of namespaces is done by enclosing one namespace declaration inside the declaration of the other namespace.

## Syntax

```
namespace Contact
{
    public class Employees
    {
        public int EmpID;
    }
    namespace Salary
    {
        public class SalaryDetails
        {
            public double EmpSalary;
        }
    }
}
```

```
namespace Bank.Accounts.EmployeeDetails
{
    public class Employees {
        public string EmpName;
    }
}
```

```
using IO = System.Console;
using Emp = Bank.Accounts.EmployeeDetails;
class AliasExample
{
    static void Main (string[] args)
    {
        Emp.Employees objEmp = new Emp.Employees();
        objEmp.EmpName = "Peter";
        IO.WriteLine("Employee Name: " + objEmp.EmpName);
    }
}
```

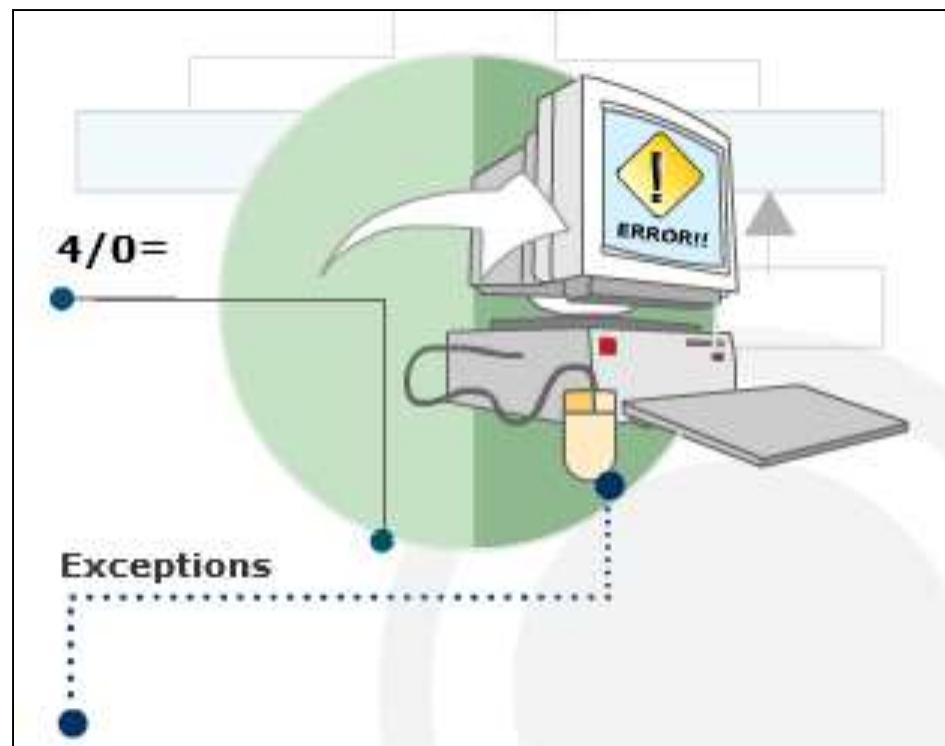
- ◆ A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- ◆ The System namespace is imported by default in the .NET Framework.
- ◆ Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- ◆ You cannot apply access modifiers such as public, protected, private, or internal to namespaces.
- ◆ A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- ◆ C# supports nested namespaces that allows you to define namespaces within a namespace.
- ◆ External aliasing in C# allows the users to define assembly qualified namespace aliases.

Session: 11

# Exception Handling

- ◆ Define and describe exceptions
- ◆ Explain the process of throwing and catching exceptions
- ◆ Explain nested **try** and multiple **catch** blocks
- ◆ Define and describe custom exceptions

- ◆ An exception is an error (abnormal event) that occurs during program execution that prevent a certain task from being completed successfully

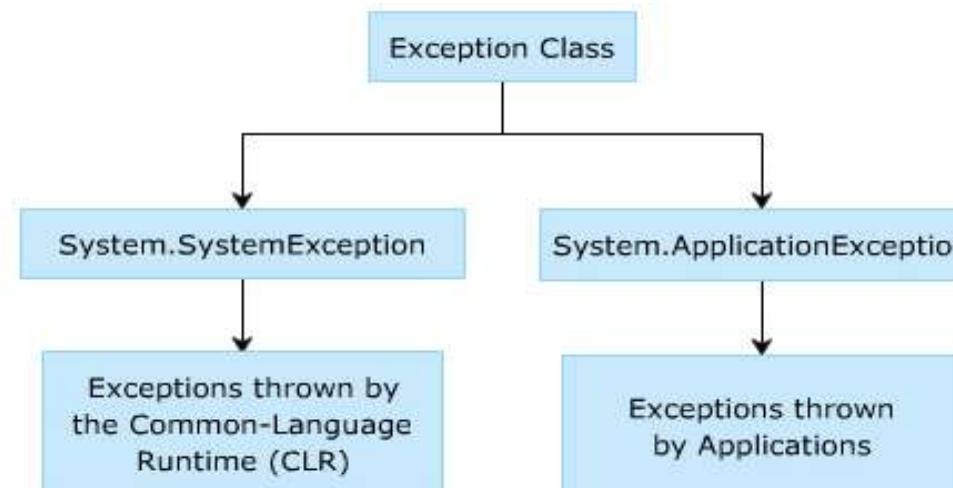


- ◆ **System-level Exceptions:**

- ◆ are thrown by the system, by the CLR.
- ◆ For example, exceptions are thrown due to failure in database connection or network connection.

- ◆ **Application-level Exceptions:**

- ◆ are thrown by user-created applications.
- ◆ For example, exceptions are thrown due to arithmetic operations or referencing any null object.



# System.Exception Class

- ◆ is the base class that handles all exceptions.

Method	Description
<b>Equals</b>	Determines whether objects are equal
<b>GetBaseException</b>	Returns a type of Exception class when overridden in a derived class
<b>GetHashCode</b>	Returns a hash function for a particular type
<b>GetObjectData</b>	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
<b>GetType</b>	Retrieves the type of the current instance
<b>ToString</b>	Returns a string representation of the thrown exception

Properties	Descriptions
<b>Message</b>	Is a message which indicates the reason for the exception.
<b>Source</b>	Provides the name of the application or the object that caused the exception.
<b>StackTrace</b>	Provides exception details on the stack at the time the exception was thrown.
<b>InnerException</b>	Returns the <code>Exception</code> instance that caused the current exception.

## Snippet

```
class ExceptionProperties {
    static void Main(string[] args) {
        byte numOne = 200;
        byte numTwo = 5;
        byte result = 0;
        try {
            result = checked((byte)(numOne * numTwo));
            Console.WriteLine("Result = {0}", result);
        }
        catch (OverflowException objEx) {
            Console.WriteLine("Message : {0}", objEx.Message);
            Console.WriteLine("Source : {0}", objEx.Source);
            Console.WriteLine("TargetSite : {0}", objEx.TargetSite);
            Console.WriteLine("StackTrace : {0}", objEx.StackTrace);
        }
        catch (Exception objEx) {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

# InvalidOperationException Class

- ◆ is thrown when an explicit conversion from a base type to another type fails.

## Snippet

```
class InvalidCastException {
    static void Main(string[] args) {
        try {
            string obj = "numOne";
            int result = (int)obj;
            Console.WriteLine("Value of numOne = {0}", result);
        }
        catch(InvalidOperationException objEx) {
            Console.WriteLine("Message : {0}", objEx.Message);
            Console.WriteLine("Error : {0}", objEx);
        }
        catch (Exception objEx) {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

# ArrayTypeMismatchException Class

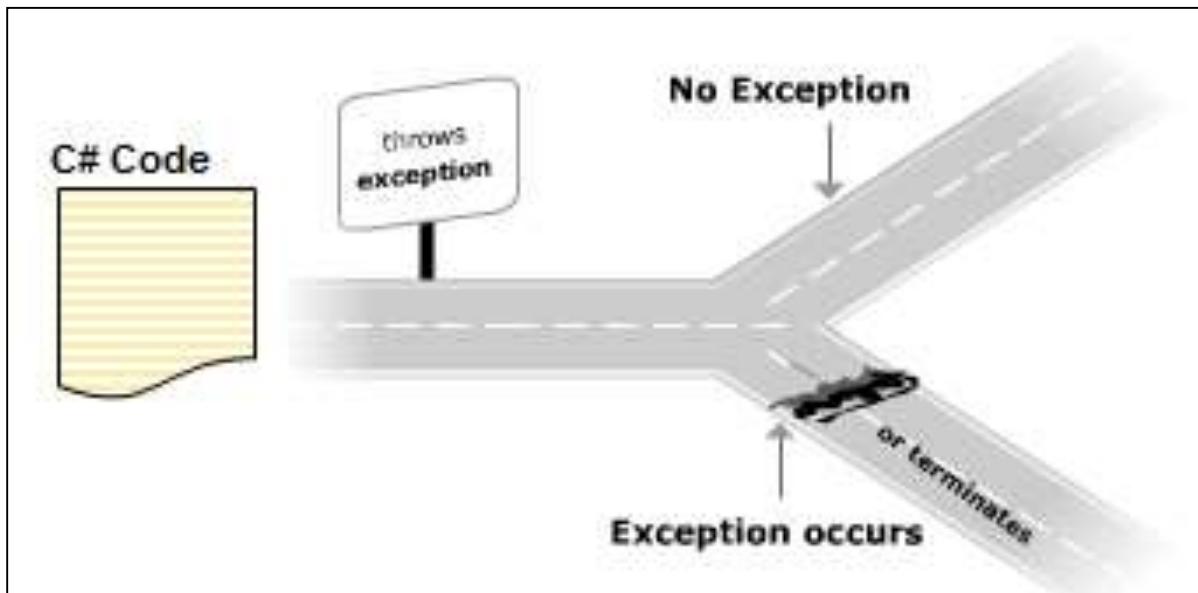
- ◆ is thrown when the data type of the value being stored is incompatible with the data type of the array.

## Snippet

```
class ArrayMisMatch {
    static void Main(string[] args){
        double[] salary = { 1000, 2000, 3000 } ;
        float[] bonus = new float[3];
        try {
            salary.CopyTo(bonus, 0);
        }
        catch (ArrayTypeMismatchException objType) {
            Console.WriteLine("Error: " + objType);
        }
        catch (Exception objEx) {
            Console.WriteLine("Error: " + objEx);
        }
    }
}
```

# Throwing and Catching Exceptions

- An exception arises when an operation cannot be completed normally. In such situations, the system throws an error.



- The error is handled through the process of **exception handling**.

- ◆ Exceptions also can be programmatically raised using the `throw` keyword.

## Syntax

```
throw exceptionObject;
```

- ◆ where,
  - ◆ `throw`: Specifies that the exception is thrown programmatically.
  - ◆ `exceptionObject`: Is an instance of a particular exception class.

- ◆ In general, if any of the statements in the `try` block raises an exception, the `catch` block is executed and the rest of the statements in the `try` block are ignored.
- ◆ Sometimes, it becomes mandatory to execute some statements irrespective of whether an exception is raised or not. In such cases, a `finally` block is used.
- ◆ The `finally` block is an optional block and it appears after the `catch` block. It encloses statements that are executed regardless of whether an exception occurs.

## Syntax

```
finally
{
    // cleanup code;
}
```

## Nested `try` and Multiple `catch` Blocks

- ◆ Exception handling code can be nested in a program. In nested exception handling, a `try` block can enclose another `try-catch` block.
- ◆ In addition, a single `try` block can have multiple `catch` blocks that are sequenced to catch and handle different type of exceptions raised in the `try` block.



- ◆ Features of the nested `try` block:

**consists of multiple (inner) try-catch constructs that starts with a `try` block, which is called the outer `try` block.**

**If an exception is thrown by a inner nested `try` block, the control passes to its corresponding nested `catch` block.**

**However, if the inner `catch` block does not contain the appropriate error handler, the control is passed to the outer `catch` block.**

# Implementing Custom Exceptions

- ◆ Custom exceptions can be created by deriving from the `Exception` class, the `SystemException` or `ApplicationException` class.

```
public class CustomError : Exception {  
    public CustomError (string message) : base(message) {}  
}  
  
public class CustomExceptionDemo {  
    static void Main(string[] args) {  
        try {  
            throw new CustomError ("This illustrates creation and catching of  
custom exception");  
        }  
        catch(CustomError ex) {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

- ◆ Exceptions are errors that encountered at run-time.
- ◆ Exception-handling allows you to handle methods that are expected to generate exceptions.
- ◆ The `try` block should enclose statements that may generate exceptions while the `catch` block should catch these exceptions.
- ◆ The `finally` block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the `try` block.
- ◆ Nested `try` blocks allow you to have a `try-catch-finally` construct within a `try` block.
- ◆ Multiple `catch` blocks can be implemented when a `try` block throws multiple types of exceptions.
- ◆ Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors.

Session: **12**

# Events, Delegates, and Collections

- ◆ Explain delegates
- ◆ Explain events
- ◆ Define and describe collections



**Delegates are objects that contain references to methods that need to be invoked.**

**Using delegates, you can call any method, which is identified only at run-time.**

**To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.**

- ◆ Declaring a delegate is quite similar to declaring a method except that there is no implementation.
- ◆ Example :

## Valid Delegate Declaration

```
public delegate int Calculation(int numOne, int numTwo);
```



## Invalid Delegate Declaration

```
public delegate Calculation(int numOne, int numTwo)  
{  
}
```



## Syntax

```
<acc_modifier> delegate <ret_type> DelegateName ([parameters]);
```

## Snippet

```
public delegate int Calculation(int numOne, int numTwo);
```

## Snippet

```
public delegate int Calculation (int numOne, int numTwo);

class Mathematics {
    static int Add(int numOne, int numTwo) {
        return (numOne + numTwo);
    }
    int Subtract(int numOne, int numTwo) {
        return (numOne - numTwo);
    }

    static void Main(string[] args) {
        int n1 = 5;
        int n2 = 23;
        Calculation oCalc = new Calculation(Add);
        Console.WriteLine("{0} + {1} = {2}", n1, n2, oCalc(n1, n2));
    }
}
```

## Output

5 + 23= 28

## Delegates & Anonymous Methods

- ◆ An anonymous method is an inline block of code that can be passed as a delegate parameter that helps to avoid creating named methods.
- ◆ The following figure displays an example of using anonymous methods:

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
    {
        Console.Write("Testing... ");
        Console.WriteLine("Threads.");
    });
    objThread.Start();
}
```

} Anonymous Method

## Snippet

```
public delegate void Maths (int valOne, int valTwo);  
  
class MathsDemo {  
    static void Add (int x, int y) {  
        Console.WriteLine("Addition: {0} + {1} = {2}",x , y, x+y);  
    }  
    static void Subtract (int x, int y) {  
        Console.WriteLine("Subtraction: {0} - {1} = {2}",x , y, x-y);  
    }  
    static void Divide(int x, int y) {  
        Console.WriteLine("Division: {0} / {1} = {2}",x , y, x/y);  
    }  
    static void Main(string[] args) {  
        Maths oMaths = new Maths (Add);  
        oMaths += new Maths (Subtract);  
        oMaths += new Maths (Divide);  
        if (oMaths != null) {      oMaths(20, 10); }  
    }  
}
```

# System.Delegate Class

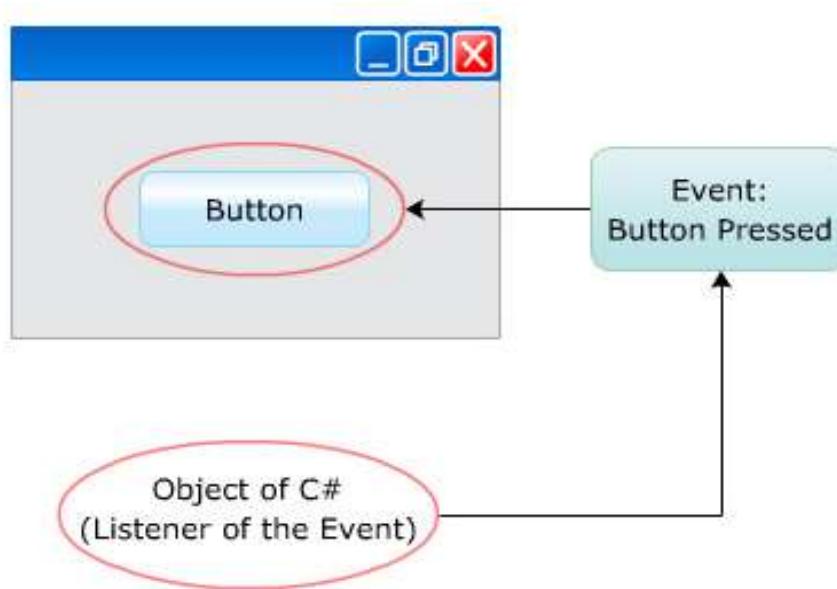
- ◆ The **Delegate** class is a built-in class defined to create delegates in C#.
- ◆ All delegates in C# implicitly inherit from the **Delegate** class.
- ◆ Constructors :

Constructor	Description
<b>Delegate(object, string)</b>	Calls a method referenced by the object of the class given as the parameter
<b>Delegate(type, string)</b>	Calls a static method of the class given as the parameter

- ◆ Properties :

Property	Description
<b>Method</b>	Retrieves the referenced method
<b>Target</b>	Retrieves the object of the class in which the delegate invokes the referenced method

- ◆ An event is a user-generated or system-generated action.
- ◆ In C#, events allow an object (source of the event) be able to notify other objects (subscribers) about the appeared event (a change having occurred).
- ◆ The following figure depicts the concept of events:



can be declared in classes and interfaces.

can be declared as abstract, virtual or sealed.

implemented using delegates.

- ◆ Events can be used to perform customized actions that are not already supported by C#.
- ◆ Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked.

- ◆ Following are the four steps for implementing events in C#:

Define a public delegate.



Create the event using the delegate.



Subscribe to listen and handle the event.



Raise the event.

- ◆ Events use delegates to call methods in objects that have subscribed to the event.
- ◆ When an event containing a number of subscribers is raised, many delegates will be invoked.

# Example

```
public delegate void PrintDetails();

class TestEvent {
    event PrintDetails Print;
    void Show()
    {
        Console.WriteLine("how to subscribe objects to an event");
    }

    static void Main(string[] args)
    {
        TestEvent o = new TestEvent();
        o.Print += new PrintDetails(o.Show);
        o.Print();
    }
}
```

- ◆ A collection is a set of related data that may not necessarily belong to the same data type that can be set or modified dynamically at run-time.
- ◆ Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers.
- ◆ However, there are differences between arrays and collections in C#:

Array	Collection
<b>Cannot be resized at run-time.</b>	<b>Can be resized at run-time.</b>
<b>The individual elements are of the <i>same</i> data type.</b>	<b>The individual elements can be of different data types.</b>
<b>Do <i>not</i> contain any methods for operations on elements.</b>	<b>Contain methods for operations on elements.</b>

# System.Collections Namespace

Class/Interface	Description
<b>ArrayList Class</b>	similar to an array except that the items can be dynamically added and retrieved from the list and it can contain values of different types
<b>Stack Class</b>	follows the Last-In-First-Out (LIFO) principle, which means the last item inserted in the collection, will be removed first
<b>Hashtable Class</b>	Provides a collection of key and value pairs that are arranged, based on the hash code of the key
<b>SortedList Class</b>	Provides a collection of key and value pairs where the items are sorted, based on the keys
<b>IDictionary Interface</b>	Represents a collection consisting of key/value pairs
<b>IDictionaryEnumerator Interface</b>	Lists the dictionary elements
<b>IEnumerable Interface</b>	Defines an enumerator to perform iteration over a collection
<b>ICollection Interface</b>	Specifies the size and synchronization methods for all collections
<b>IEnumerator Interface</b>	Supports iteration over the elements of the collection
<b>IList Interface</b>	Represents a collection of items that can be accessed by their index number

- ◆ Features of **ArrayList**:

is a variable-length array.  
can store elements of different data types.  
can accept null values.  
can also include duplicate elements.

allows to specify the size and the capacity of the collection.  
default capacity is 16.

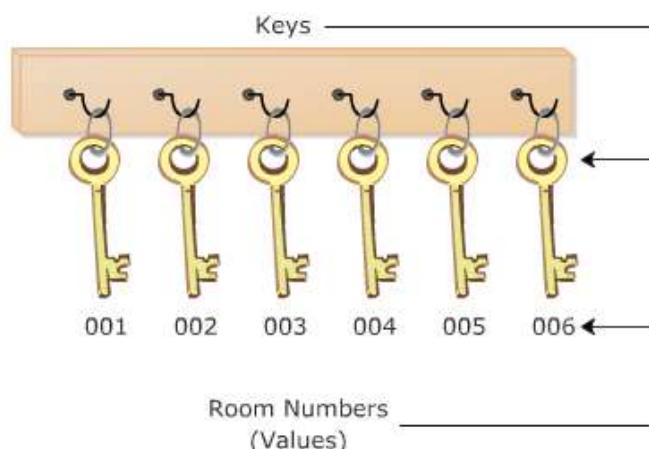
If the number of elements in the list reaches the specified capacity, the capacity of the list gets doubled automatically..

consists of different methods and properties that are used to add, modify, and delete any element in the list even at run-time.

can be accessed the elements by using the index position.

## Hashtable Class

- ◆ Consider the reception area of a hotel where you find the keyholder storing a bunch of keys.
- ◆ Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.
- ◆ Similar to the keyholder, the **Hashtable** class in C# allows you to create collections in the form of keys and values that associates keys with their corresponding values.
- ◆ The **Hashtable** class uses the hashtable to retrieve values associated with their unique key.



- ◆ represents a collection of key and value pairs where elements are sorted according to the key.
- ◆ By default, the **SortedList** sorts the elements in ascending order, however, this can be changed if an **IComparable** object is passed to the constructor of the **SortedList** class.
- ◆ These elements are either accessed using the corresponding keys or the index numbers.
- ◆ If you access elements using their keys, the **SortedList** behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.

- ◆ consists of a generic collection of elements organized in key and value pairs and maps the keys to their corresponding values.
- ◆ The following syntax declares a **Dictionary** generic class:

```
Dictionary<TKey, TValue>
```

- ◆ A delegate in C# is used to refer to a method in a safe manner.
- ◆ An event is a data member that enables an object to provide notifications to other objects about a particular action.
- ◆ The **System.Collections.Generic** namespace consists of generic collections that allow reusability of code and provide better type-safety.
- ◆ The **ArrayList** class allows you to increase or decrease the size of the collection during program **execution**.
- ◆ The **Hashtable** class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the hashtable is uniquely identified by its key.
- ◆ The **SortedList** class allows you to store elements as key and value pairs where the data is sorted based on the key.
- ◆ The **Dictionary** generic class represents a collection of elements organized in key and value pairs.

Session: 13

# Generics and Iterators

- ◆ Define and describe generics
- ◆ Explain creating and using generics
- ◆ Explain iterators



## Example

Consider a program that uses an array variable of type **Object** to store a collection of student names.

The names are read from the console and stored as type Object.

If you enter numeric data, it will be accepted without any verification because it allows to cast any value to and from Object !

- ◆ To ensure type-safety, C# introduces generics, which has a number of features including the ability to allow you to define generalized type templates based on which the type can be constructed later.
- ◆ *Generics are a kind of **parameterized** data structures that can work with value types as well as reference types.*

- ◆ ensure type-safety at compile-time:
  - ◆ ensure strongly-typed programming model
- ◆ allow to reuse the code in a safe manner without casting or boxing:
  - ◆ reduce run-time errors
  - ◆ Improve performance because of low memory usage as no casting or boxing operation is required.
- ◆ can be reusable with different types but can accept values of a single type at a time.

# Namespaces, Classes, and Interfaces for Generics

- ◆ There are several namespaces in the .NET Framework that facilitate creation and use of generics which are as follows:

**System.Collections.ObjectModel**

- consists of classes and interfaces that allow you to create type-safe collections.

**System.Collections.Generic**

- consists of classes and interfaces that can be used as collections in the object model of a reusable library

- ◆ are declared with **type parameters** enclosed within angular brackets.
- ◆ can apply some restrictions or constraints to the type parameters by using the **where** keyword.

## Syntax

```
<acc_modifier> class <ClassName><<type-parameter list>> [where <type-parameter constraint clause>]
```

- ◆ where,
  - ❖ **type-parameter list:** Is used as a placeholder for the actual data type.
  - ❖ **type-parameter constraint clause:** Is an option applied to the type parameter with the **where** keyword.
- ◆ Example:
  - ❖ **public class Student<T> { ... }**

## Constraints on Type Parameters

- ◆ A constraint is a restriction imposed on the data type of the type parameter and are specified using the **where** keyword.
- ◆ Types of constraints that can be applied to the type parameter:

Constraints	Descriptions of Type parameter
<b>T : struct</b>	must be of a value type only except the null value
<b>T : class</b>	must be of a reference type such as a class, interface, or a delegate
<b>T : new()</b>	must consist of a constructor without any parameter which can be invoked publicly
<b>T : &lt;base class name&gt;</b>	must be the parent class or should inherit from a parent class
<b>T : &lt;interface name&gt;</b>	must be an interface or should inherit an interface

# Inheriting Generic Classes

- ◆ inherit a generic class from an existing generic class:

## Syntax

```
<acc_modifier> class <BaseClass><<type-parameter>>{ }  
<acc_modifier> class <DerivedClass>:<BaseClass><<type-parameter>>{ }
```

where,

- ◆ <**type-parameter**>: Is a placeholder for the specified data type.
- ◆ **DerivedClass**: Is the generic derived class.

- ◆ inherit a non-generic class from a generic class:

## Syntax

```
<acc_modifier> class <BaseClass><<type-parameter>>{ }  
<acc_modifier> class <DerivedClass>:<BaseClass><<type-par-value>>{ }
```

where,

- ◆ <**type-par-value**>: Can be a data type such as int, string, or float.

- ◆ process values whose data types are known only when accessing the variables that store these values.
- ◆ declared with the type-parameter list enclosed within angular brackets.
- ◆ allow to call the method with a different type.
- ◆ can be declared within generic or non-generic class.
- ◆ When declared within a generic class, the body of the method refers to the type parameters of both, the method and class declaration.
- ◆ can be declared with the following keywords:
  - ◆ **Virtual**.
  - ◆ **Override**: while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.
  - ◆ **Abstract**

## Syntax

```
acc_modifier interface InterfaceName <type-parameter list>  
[where <type-parameter constraint clause>]
```

where,

- ❖ **Type-parameter constraint clause:** Is an optional class or an interface applied to the type parameter with the **where** keyword.

- ◆ Delegates are reference types that encapsulate a reference to a method that has the same signature and return type.
- ◆ Features of a generic delegate:
  - ◆ can be used to refer to multiple methods in a class with different types of parameters.
  - ◆ The number of parameters of the delegate and the referenced methods must be the same.
  - ◆ The type parameter list is specified after the delegate's name in the syntax.

```
delegate <return_type><DelegateName><type  
parameter list>(<argument_list>);
```

- ◆ where,
  - ◆ **return\_type:** Determines the type of value the delegate will return.
  - ◆ **DelegateName:** Is the name of the generic delegate.
  - ◆ **type parameter list:** Is used as a placeholder for the actual data type.
  - ◆ **argument\_list:** Specifies the parameter within the delegate.

# Overloading Methods Using Type Parameters

- ◆ can overload methods of a generic class that take generic type parameters by changing the type or the number of parameters.
- ◆ However, the type difference is not based on the generic type parameter, but is based on the data type of the parameter passed.

```
class General<T, U>{
    T _valOne;
    U _valTwo;
    public void AcceptValues(T item) {
        _valOne = item;
    }
    public void AcceptValues(U item) {
        _valTwo = item;
    }
    public void Display() {
        Console.WriteLine(_valOne + "\t" + _valTwo);
    }
}
```

# Overriding Virtual Methods in Generic Class

- ◆ Can override methods in generic classes by using keywords **virtual** and **override**.

## Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;

class GeneralList<T>
{
    protected T ItemOne;

    public GeneralList(T valOne) {
        ItemOne = valOne;
    }

    public virtual T GetValue() {
        return ItemOne;
    }
}
```

```
class Student<T>:GeneralList<T> {
    public T Value;

    public Student(T val1, T val2):
        base(val1) {
            Value = val2;
    }

    public override T GetValue() {
        Console.WriteLine(base.GetValue());
        return Value;
    }
}
```

- ◆ For a class that behaves like a collection, it is preferable to use iterators to iterate through the values of the collection with the **foreach** statement.
- ◆ Benefits of iterator:
  - ◆ provide a simplified and faster way of iterating through the values of a collection.
  - ◆ reduce the complexity of providing an enumerator for a collection.
  - ◆ can return large number of values.
  - ◆ can be used to evaluate and return only those values that are needed.
  - ◆ can return values without consuming memory by referring each value in the list.

# Implementation Iterators

- ◆ Iterators can be created by implementing interface **IEnumerable** :  
**override** method **IEnumerator GetEnumerator()**
- ◆ The iterator block uses :
  - ◆ **yield return** to provide values to the instance of the enumerator
  - ◆ **yield break** to terminate the iteration process.

```
class Department : IEnumerable
{
    string[] departmentNames = {"Marketing", "Finance",
    "Information Technology", "Human Resources"};
    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < departmentNames.Length; i++)
        {
            yield return departmentNames[i];
        }
    }
}
```

```
static void Main (string [] args)
{
    Department objDepartment = new Department();
    Console.WriteLine("Department Names");
    Console.WriteLine();
    foreach(string str in objDepartment)
    {
        Console.WriteLine(str);
    }
}
```

## Generic Iterators

- ◆ Generic iterators are created by method **GetEnumerator()** returning an object of the generic **IEnumerator<T>** or **IEnumerable<T>** interface.
- ◆ They are used to iterate through values of any value type.
- ◆ For example

```
using System;
using System.Collections.Generic;
class GenericDepartment<T>
{
    T[] item;
    public GenericDepartment(T[] val)
    {
        item = val;
    }
    public IEnumerator<T> GetEnumerator()
    {
        foreach (T value in item)
        {
            yield return value;
        }
    }
}
```

```
class GenericIterator
{
    static void Main(string[] args)
    {
        string[] departmentNames = { "Marketing", "Finance",
            "Information Technology", "Human Resources" };
        GenericDepartment<string> objGeneralName = new
        GenericDepartment<string>(departmentNames);
        foreach (string val in objGeneralName)
        {
            Console.Write(val + "\t");
        }
        int[] departmentID = { 101, 110, 210, 220 };
        GenericDepartment<int> objGeneralID = new
        GenericDepartment<int>(departmentID);
        Console.WriteLine();
        foreach (int val in objGeneralID)
        {
            Console.Write(val + "\t\t");
        }
        Console.WriteLine();
    }
}
```

- ◆ Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- ◆ Generics provide several benefits such as type-safety and better performance.
- ◆ Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- ◆ Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- ◆ An iterator is a block of code that returns sequentially ordered values of the same type.
- ◆ One of the ways to create iterators is by using the GetEnumerator() method of the IEnumerable or IEnumerator interface.
- ◆ The yield keyword provides values to the enumerator object or to signal the end of the iteration.

Session: 14

# Advanced Methods and Types

- ◆ Describe anonymous methods
- ◆ Define extension methods
- ◆ Explain anonymous types
- ◆ Explain partial types
- ◆ Explain nullable types

- An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.

## Example

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
    {
        Console.Write("Testing... ");
        Console.WriteLine("Threads.");
    });
    objThread.Start();
}
```

} Anonymous Method

# Creating Anonymous Methods

- ◆ Define a delegate :

```
<acc-modifier> delegate <ret-type> Delegate_name(parameters) ;
```

- ◆ Instantiate the delegate by using anonymous method:

```
<Delegate_name> obj = delegate(parameters) {  
    /* . . . */  
}
```

```
class AnonymousMethods {  
    delegate void Display();  
  
    static void Main(string[] args) {  
        //using anonymous methods  
        Display objDisp = delegate() {  
            Console.WriteLine("This is an anonymous method");  
        } ;  
        objDisp();  
    }  
}
```

Snippet

# Referencing Multiple Anonymous Methods

- ◆ C# allows a delegate that can reference multiple anonymous methods.
- ◆ The `+ =` operator is used to add additional references to either named or anonymous methods after instantiating the delegate.

```
class Program
{
    public delegate void AnoMethod(int a, int b);
    static void Main(string[] args)
    {
        int a = 9, b = 3;
        AnoMethod add ;

        add = delegate(int x, int y)  {
            Console.WriteLine("{0} + {1} = {2}", x, y, x + y);
        };

        add += delegate(int x, int y)  {
            Console.WriteLine("{0} - {1} = {2}", x, y, x - y);
        };
        add(a, b);
    }
}
```

- ◆ allow to extend an existing type with new functionality without directly modifying those types.
- ◆ are **static** methods that have to be declared in a **static** class.
- ◆ declared by specifying the first parameter with the **this** keyword, identifies the type of objects in which the method can be called.
- ◆ The object that you use to invoke the method is automatically passed as the first parameter.

## Syntax

```
static return-type MethodName (this type-obj, param-list)
```

- The following code creates an extension method for a string and converts the first character of the string to lowercase:

## Snippet

```
using System;
static class ExtensionExample
{
    // Extension Method to convert the 1st char to lowercase
    public static string FirstLetterLower(this string result)
    {
        if (result.Length > 0){
            char[] s = result.ToCharArray();
            s[0] = char.ToLower(s[0]);
            return new string(s);
        }
        return result;
    }
}
```

- ◆ Anonymous type:

- ◆ Is basically a class with no name and is not explicitly defined in code.
- ◆ Uses object initializers to initialize properties and fields. Since it has no name, you need to declare an implicitly typed variable to refer to it.

## Syntax

```
new { identifierA = valueA, identifierB = valueB, ... }
```

```
using System;
/// <summary>
/// Class AnonymousTypeExample to demonstrate anonymous type
/// </summary>
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michigan Enterprises", Code = 1301,
                         Price = 35056.75 };
        Console.WriteLine("Stock Name: " + stock.Name);
        Console.WriteLine("Stock Code: " + stock.Code);
        Console.WriteLine("Stock Price: " + stock.Price);
    }
}
```

- ◆ A large project in an organization involves creation of multiple structures, classes, and interfaces.
- ◆ If these types are stored in a single file, their modification and maintenance becomes very difficult.
- ◆ In addition, multiple programmers working on the project cannot use the file at the same time for modification.
- ◆ Thus, partial types can be used to split a type over separate files, allowing the programmers to work on them simultaneously.
- ◆ Partial types are also used with the code generator in Visual Studio 2012.

- ◆ facilitates the definition of classes, structures, and interfaces over multiple files.
- ◆ Benefits of Partial Type
  - ◆ separate the generator code from the application code.
  - ◆ help in easier development and maintenance of the code.
  - ◆ make the debugging process easier.
  - ◆ prevent programmers from accidentally modifying the existing code.
- ◆ The following figure displays an example of a partial type:

**File 1**

```
partial struct Sample
{
    <MethodOne>;
}
```

**File 2**

```
partial struct Sample
{
    <MethodTwo>;
}
```

## Merged Elements during Compilation

- ◆ The members of partial classes, partial structures, partial interfaces declared & stored at different locations are combined together at the time of compilation.
- ◆ These members can include:
  - ❖ XML comments & Interfaces
  - ❖ Generic-type parameters & Class variables
  - ❖ Local variables & Methods
  - ❖ Properties
- ◆ A partial type can be compiled at the Developer Command Prompt for VS2012.  
The command to compile a partial type is:

**csc /out:<FileName>.exe <CSFileName1>.cs <CSFileName2>.cs**

```
D:\C#>csc /out:StudentInfo.exe StudentDetails.cs Students.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
D:\C#>StudentInfo
Student Roll Number: 20
Student Name: Frank
```

- ◆ is a method whose signature is included in a partial type.
- ◆ may be optionally implemented in another part of the partial class or type or same part of the class or type.

## Snippet

```
namespace PartialTest
{
    /// <summary>
    /// Class Shape is a partial class and defines a partial method.
    /// </summary>

    public partial class Shape
    {
        partial void Create();
    }
}
```

- ◆ A nullable type can include any range of values that is valid for the data type to which the nullable type belongs.
- ◆ For example, a bool type that is declared as a nullable type can be assigned the values true, false, or null.
- ◆ Nullable types have two public read-only properties that can be implemented to check the validity of nullable types and to retrieve their values.

These are as follows:

- ◆ **The HasValue property:** **HasValue** is a **bool** property that determines validity of the value in a variable.  
The **HasValue** property returns a true if the value of the variable is **not null**, else it returns false.
- ◆ **The Value property:** The **Value** property identifies the value in a nullable variable. When the **HasValue** evaluates to true, the **Value** property returns the value of the variable, otherwise it returns an exception.

## Implementing Nullable Types 2-2

- The following code displays the employee's name, ID, and role using the nullable types:

```
using System;
class Employee {
    static void Main(string[] args)  {
        int empId = 10;
        string empName = "Patrick";
        char? role = null;
        Console.WriteLine("Employee ID: " + empId);
        Console.WriteLine("Employee Name: " + empName);
        if (role.HasValue == true) {
            Console.WriteLine("Role: " + role.Value);
        }
        else {
            Console.WriteLine("Role: null");
        }
    }
}
```

Snippet

- When a nullable type contains a null value and you assign this nullable type to a non-nullable type, the compiler generates an exception called **System.InvalidOperationException**.
- To avoid this problem, you can specify a default value for the nullable type that can be assigned to a non-nullable type by using the **??** operator.
- If the nullable type contains a null value, the **??** operator returns the default value.
- The following code demonstrates the use of **??** operator:

```
using System;
class Salary {
    static void Main(string[] args) {
        double? actualValue = null;
        double marketValue = actualValue ?? 0.0;
        actualValue = 100.20;
        Console.WriteLine("Value: " + actualValue);
        Console.WriteLine("Market Value: " + marketValue);
    }
}
```

Snippet

- ◆ Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- ◆ Extension methods allow you to extend different types with additional static methods.
- ◆ You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- ◆ Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- ◆ You can define partial types using the partial keyword.
- ◆ Nullable types allow you to assign null values to the value types.
- ◆ Nullable types provide two public read-only properties, HasValue and Value.

Session: 15

# Advanced Concepts of C#

- ◆ Describe system-defined generic delegates
- ◆ Define lambda expressions
- ◆ Explain query expressions LinQ
- ◆ Describe Windows Communication Framework (WCF)
- ◆ Explain parallel programming
- ◆ Explain dynamic programming

# System-Defined Generic Delegates

- ◆ Following are the commonly used predefined generic delegates:

`Func<TResult>()`  
Delegate

represents a method having zero parameters and returns a value of type **TResult**.

`Func<T, TResult>(T arg)`  
Delegate

represents a method having one parameter of type **T** and returns a value of type **TResult**.

`Func<T1, T2, TResult>(T1 arg1, T2 arg2)` Delegate

represents a method having two parameters of type **T1** and **T2** respectively and returns a value of type **TResult**.

```
public class WordLength{
    public static void Main() {
        Func<string, int> cntWord = Count;
        string location = "Netherlands";
        // Use delegate instance to call Count method
        Console.WriteLine("The number of characters in the input is:
{0}", cntword(location).ToString());
    }
    private static int Count(string inputString) {
        return inputString.Length;
    }
}
```

- ◆ A method associated with a delegate is never invoked by itself, instead, it is only invoked through the delegate.

## Syntax

```
parameter-list => expression or statements
```

- ◆ where,
  - ❖ **parameter-list** : is an explicitly typed or implicitly typed parameter list
  - ❖ **=>** : is the lambda operator

## Example

```
class Program {
    delegate int ProcessNumber(int input);

    static void Main(string[] args) {
        ProcessNumber square = n => n * n;
        Console.WriteLine(square(5));
    }
}
```

# Lambdas with Standard Query Operators

- ◆ Lambda expressions can also be used with standard query operators.

Operator	Description
Sum	Calculates sum of the elements in the expression
Count	Counts the number of elements in the expression
OrderBy	Sorts the elements in the expression
Contains	Determines if a given value is present in the expression

- ◆ Example

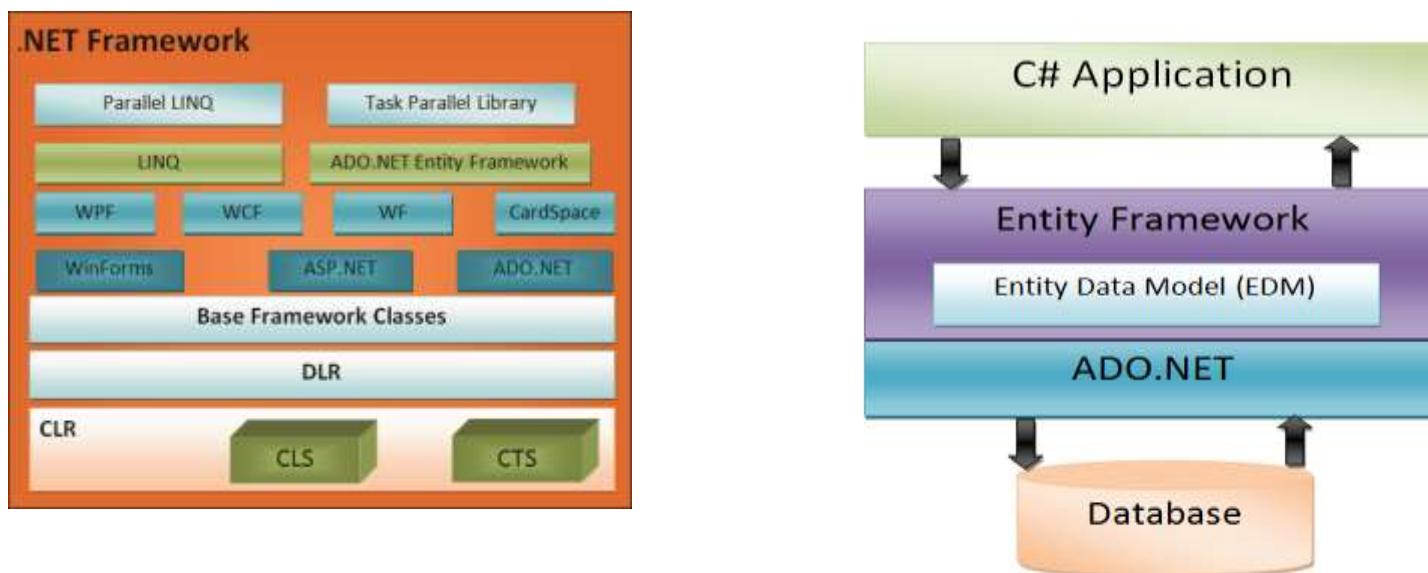
```
public class NameSort{
    public static void Main() {
        // Declare and initialize an array of strings
        string[] names={"Hanna","Jim","Peter","Karl","Abby"};
        Foreach (string item in names.OrderBy(s => s)) {
            Console.WriteLine(item);
        }
    }
}
```

- ◆ A query expression is a query that is written in syntax using clauses such as **from**, **select**... These clauses are an inherent part of a LINQ query.
- ◆ LINQ is introduced in Visual Studio 2008 that simplifies working with data present in various formats in different data sources.
- ◆ A **from** clause must be used to start a query expression and a **select** or **group** clause must be used to end the query expression.

Clause
from
where
select
group
orderby
ascending
descending

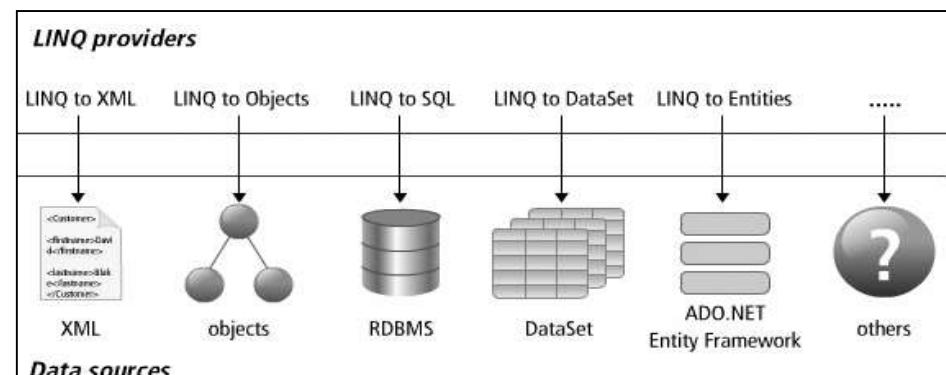
```
class Program {
    static void Main(string[] args) {
        string[] names = {"Hanna", "Jim", "Pearl", "Mel",
                          "Jill", "Peter", "Karl", "Abby", "Benjamin" };
        IEnumerable<string> items = from word in names
                                      where word.EndsWith("l")
                                      select word;
        foreach (string s in items) {
            Console.WriteLine(s);
        }
    }
}
```

- ◆ The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- ◆ EDM allows to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it.

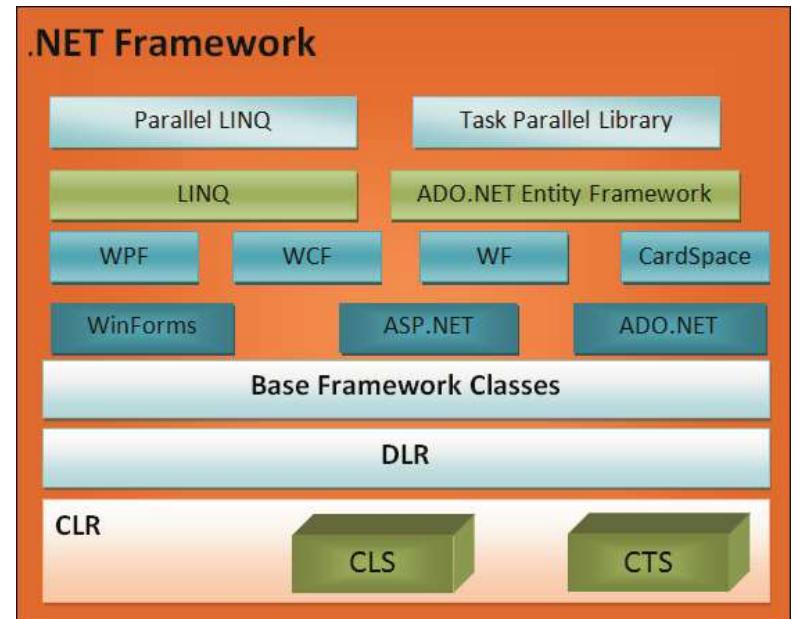


# Querying Data by Using LINQ Query Expressions

- ◆ LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources.
- ◆ However, different data sources accept queries in different formats. To solve this problem, LINQ provides the various LINQ providers, such as LINQ to Entities, LINQ to SQL, LINQ to Objects, and LINQ to XML...
- ◆ To create a query, needs a data source against which the query will execute. An instance of the ObjectQuery class represents the data source.
- ◆ In LINQ to entities, a query is stored in a variable. When the query is executed, it is first converted into a command tree representation that is compatible with the Entity Framework.
- ◆ Then, the Entity Framework executes the query against the data source and returns the result.



- ◆ Windows Communication Foundation (WCF) is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- ◆ SOA is an extension of distributed computing based on the request/response design pattern.
- ◆ SOA allows creating interoperable services that can be accessed from heterogeneous systems.
- ◆ Interoperability enables a service provider to host a service on any hardware or software platform that can be different from the platform on the consumer end.



# The System.Threading.Thread Class

- ◆ allows to create and control a thread in a multithreaded application.
- ◆ Each thread passes through different states that are represented by the members of the **ThreadState** enumeration.
- ◆ A new thread can be instantiated by passing to the constructor of the Thread class a **ThreadStart** delegate that represents the method that the new thread will execute.
- ◆ Once a thread is instantiated, it can be started by making a call to the Start() method.

Snippet

```
class ThreadDemo {  
    public static void Print() {  
        while (true) Console.Write("1");  
    }  
    static void Main (string [] args) {  
        Thread newThread = new Thread(new ThreadStart(Print));  
        newThread.Start();  
        while (true) Console.Write("2");  
    }  
}
```

- ◆ The generic collection classes provides improved type safety and performance. However, they are not thread safe.
- ◆ To address the problems, the .NET Framework provides concurrent collection classes in the System.Collections.Concurrent namespace.
- ◆ These classes relieves programmers from providing thread synchronization code when multiple threads simultaneously accesses these collections.
- ◆ The important classes :

ConcurrentDictionary  
< TKey, TValue >

- Is a thread-safe implementation of a dictionary of key-value pairs.

ConcurrentQueue<T>

- Is a thread-safe implementation of a queue.

ConcurrentStack<T>

- Is a thread-safe implementation of a stack.

ConcurrentBag<T>

- Is a thread-safe implementation of an unordered collection of elements.

# The System.Threading.Task Class

- ◆ TPL (Task Parallel Library) provides the Task class represents an asynchronous task in a program.
- ◆ Task is created by providing a user delegate that encapsulates the code that the task will execute. The delegate can be a named delegate, such as the Action delegate, an anonymous method, or a lambda expression. After that, calls the Start() method to start the task.
- ◆ This method passes the task to the task scheduler that assigns threads to perform the work.
- ◆ To ensure that a task completes before the main thread exits, call the Wait() method of the Task class.
- ◆ To ensure that all the tasks in the array of the Tasks object complete, call the AllComplete() method of the Task class.
- ◆ The Task class also provides the TryInvoke() method for performing an operation.

## Snippet

```
class TaskDemo {  
    private static void printMessage() {  
        Console.WriteLine("Executed by a Task");  
    }  
  
    static void Main (string [] args) {  
        Task t1 = new Task(new Action(printMessage));  
        t1.Start();  
        Task t2 = Task.Run(() => printMessage());  
        t1.Wait();  
        t2.Wait();  
        Console.WriteLine("Exiting main method");  
    }  
}
```

- ◆ TPL provides support for asynchronous programming through two new keywords: `async` and `await`.
- ◆ These keywords can be used to asynchronously invoke long running methods in a program.
- ◆ A method marked with the `async` keyword notifies the compiler that the method will contain at least one `await` keyword.
- ◆ If the compiler finds a method marked as `async` but without an `await` keyword, it reports a compilation error.
- ◆ The `await` keyword is applied to an operation to temporarily stop the execution of the `async` method until the operation completes.
- ◆ In the meantime, control returns to the `async` method's caller. Once the operation marked with `await` completes, execution resumes in the `async` method.
- ◆ A method marked with the `async` keyword can have either one of the following return types:
  - ❖ `void`
  - ❖ `Task`
  - ❖ `Task<TResult>`

- ◆ LINQ to Object refers to the use of LINQ queries with enumerable collections, such as List<T> or arrays.
- ◆ PLINQ is the parallel implementation of LINQ to Object. While LINQ to Object sequentially accesses an in-memory IEnumerable or IEnumerable<T> data source, PLINQ attempts parallel access to the data source based on the number of processor in the host computer.
- ◆ For parallel access, PLINQ partitions the data source into segments, and then executes each segment through separate threads in parallel.
- ◆ The System.Linq.ParallelEnumerable provides methods that implement PLINQ functionality.

```
string[] arr = new string[] { "Peter", "Sam", "Philip", "Andy", "Philip",
    "Mary", "John", "Pamela" };
var query = from string name in arr select name;
Console.WriteLine("Names retrieved using sequential LINQ");
foreach (var n in query) {
    Console.WriteLine(n);
}

var plinqQuery = from string name in arr.AsParallel() select name;
Console.WriteLine("Names retrieved using PLINQ");
foreach (var n in plinqQuery) {
    Console.WriteLine(n);
}
```

- ◆ C# provides dynamic types to support dynamic programming for interoperability of .NET applications with dynamic languages such as IronPython and COM APIs such as the Office Automation APIs.
- ◆ The C# compiler does not perform static type checking on objects of a dynamic type. The type of a dynamic object is resolved at runtime using the Dynamic Language Runtime (DLR).
- ◆ A programmer using a dynamic type is not required to determine the source of the object's value during application development.
- ◆ However, any error that escapes compilation checks causes a run-time exception.

## Snippet

```
class DemoClass {
    public void Operation(String name)  {
        Console.WriteLine("Hello {0}", name);
    }
}
class DynamicDemo {
    static void Main(string[] args)  {
        dynamic dynaObj = new DemoClass();
        dynaObj.Operation();
    }
}
```

- ◆ System-defined generic delegates take a number of parameters of specific types and return values of another type.
- ◆ A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.
- ◆ A query expression is a query that is written in query syntax using clauses such as from, select, and so forth.
- ◆ The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- ◆ WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- ◆ Various classes and interfaces in the `System.Threading` namespace provide built-in support for multithreaded programming in the .NET Framework.
- ◆ To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.

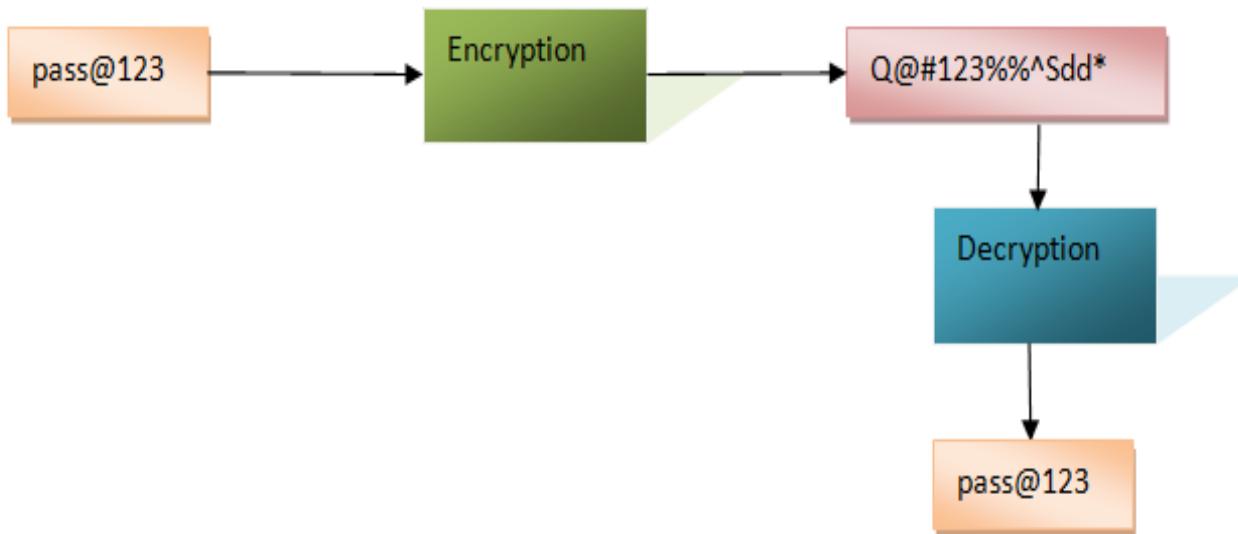
Session: **16**

# Encrypting and Decrypting Data

- ◆ Explain symmetric encryption.
- ◆ Explain asymmetric encryption.
- ◆ List the various types in the **System.Security.Cryptography** namespace that supports symmetric and asymmetric encryptions.

- ❖ All organizations need to handle sensitive data that can be either present in storage or may be exchanged between different entities within and outside the organization over a network.
- ❖ Such data is often prone to misuse either intentionally with malicious intent or unintentionally.
- ❖ To avoid such misuse, there should be some security mechanism that can ensure confidentiality and integrity of data.

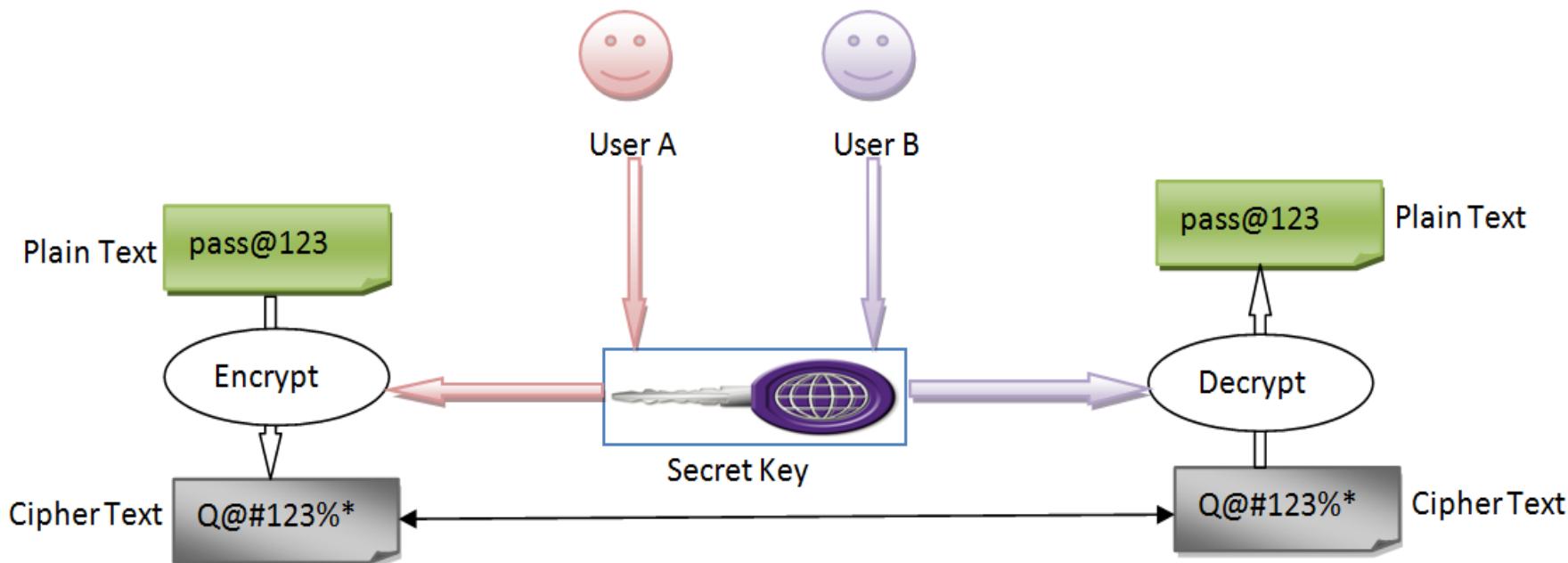
- The following figure shows the process of encryption and decryption of a password as an example.



- In the figure:
  - The plain text, pass@123 is encrypted to a cipher text.
  - The cipher text is decrypted back to the original plain text.

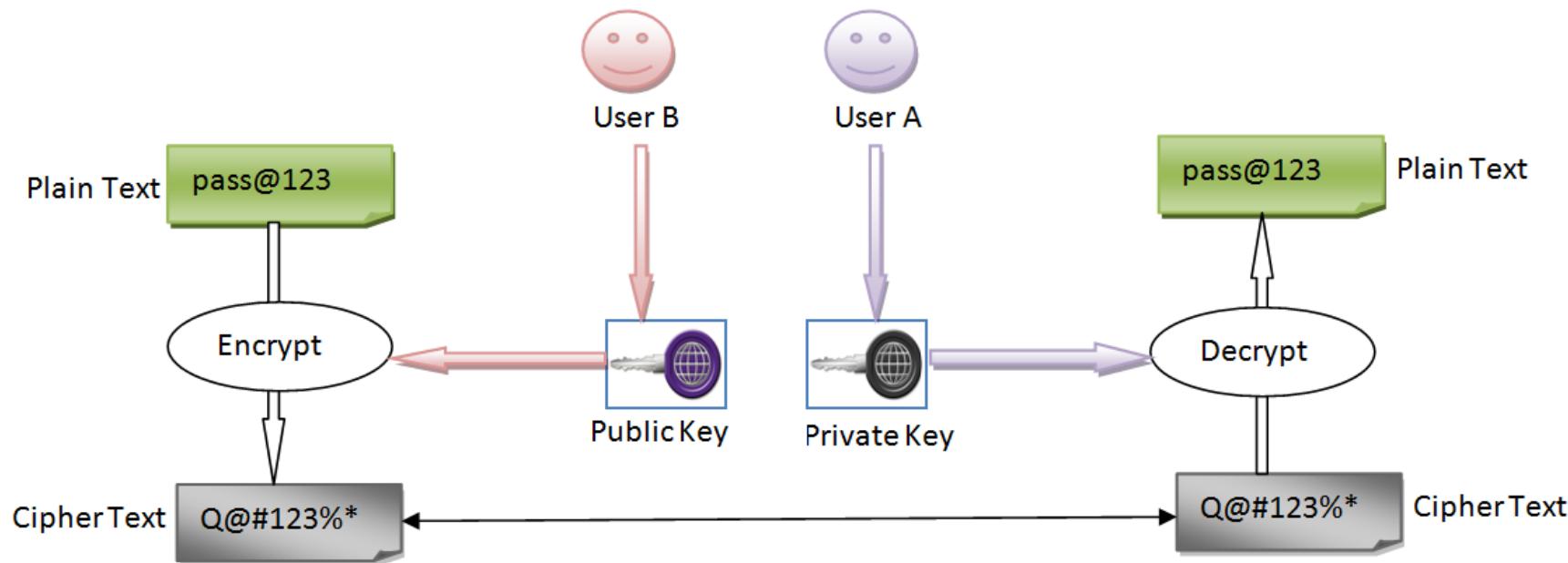
# Symmetric Encryption

- ◆ Symmetric encryption, or secret key encryption, uses a single key, known as the secret key both to encrypt and decrypt data.
  - ❖ User A uses a secret key to encrypt a plain text to cipher text.
  - ❖ User A shares the cipher text and the secret key with User B.
  - ❖ User B uses the secret key to decrypt the cipher text back to the original plain text.



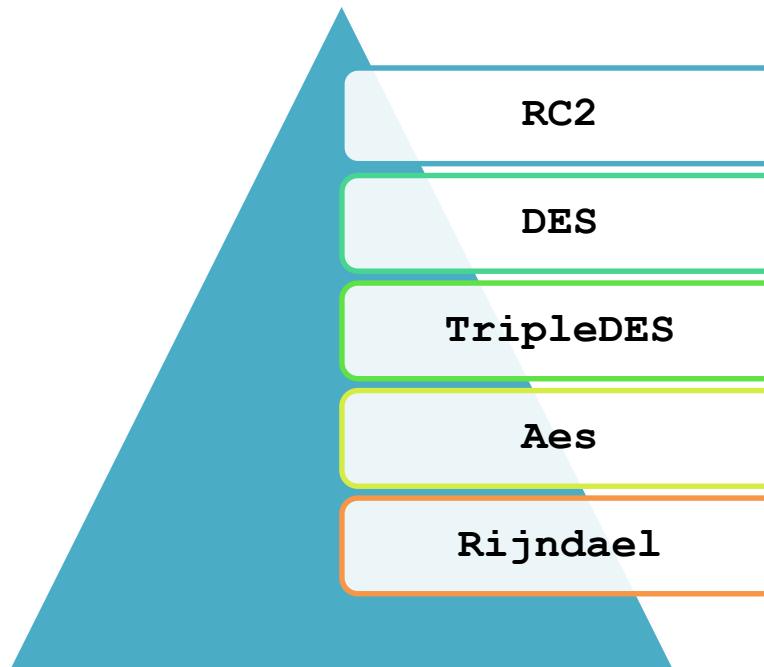
# Asymmetric Encryption

- ◆ Asymmetric encryption uses a pair of public and private key to encrypt and decrypt data.
- ◆ User A generates a public and private key pair.
- ◆ User A shares the public key with User B.
- ◆ User B uses the public key to encrypt a plain text to cipher text.
- ◆ User A uses the private key to decrypt the cipher text back to the original plain text.



# Symmetric Encryption Algorithms 1-6

- ◆ The **System.Security.Cryptography** namespace provides the **SymmetricAlgorithm** base class for all symmetric algorithm classes.
- ◆ The derived classes of the **SymmetricAlgorithm** base class are as follows:



## RC2

- ◆ Is an abstract base class for all classes that implement the RC2 algorithm.
- ◆ Is a proprietary algorithm developed by RSA Data Security, Inc. in 1987.
- ◆ Supports key sizes ranging from **40 bits to 128** bits in 8-bit increments for encryption.
- ◆ Was designed for the old generation processors and currently have been replaced by more faster and secure algorithms.
- ◆ Derives the **RC2CryptoServiceProvider** class to provide an implementation of the RC2 algorithm.

## DES

- ◆ Is an abstract base class for all classes that implement the Data Encryption Standard (DES) algorithm.
- ◆ Developed by IBM but as of today available as a U.S. Government Federal Information Processing Standard (FIPS 46-3).
- ◆ Works on the data to encrypt as blocks where each block is of 64 bits.
- ◆ Uses a **key of 64** bits to perform the encryption. On account of its small key size DES encrypts data faster as compared to other symmetric algorithms.
- ◆ Is prone to brute force security attacks because of its smaller key size.
- ◆ Derives the **DESCryptoServiceProvider** class to provide an implementation of the DES algorithm.

## TripleDES

- ◆ Is an abstract base class for all the classes that implement the TripleDES algorithm.
- ◆ Is an enhancement to the DES algorithm for the purpose of making the DES algorithm more secure against security threats.
- ◆ Works on 64 bit blocks that is similar to the DES algorithm.
- ◆ Supports key sizes of **128 bits to 192 bits**.
- ◆ Derives the **TripleDESCryptoServiceProvider** class to provide an implementation of the TripleDES algorithm.

## Aes

- ◆ Is an abstract base class for all classes that implement the Advanced Encryption Standard (AES) algorithm.
- ◆ Is a successor of DES and is currently considered as one of the most secure algorithm.
- ◆ Is more efficient in encrypting large volume of data in the size of several gigabytes.
- ◆ Works on 128-bits blocks of data and key sizes of **128, 192, or 256** bits for encryption.
- ◆ Derives **the AesCryptoServiceProvider** and **AesManaged** classes to provide an implementation of the AES algorithm.

## Rijndael

- ◆ Is an abstract base class for all the classes that implement the Rijndael algorithm.
- ◆ Is a superset of the Aes algorithm.
- ◆ Supports key sizes of **128, 192, or 256** bits similar to the Aes algorithm.
- ◆ Can have block sizes of 128, 192, or 256 bits, unlike Aes, which has a fixed block size of 128 bits.
- ◆ Provides the flexibility to select an appropriate block size based on the volume of data to encrypt by supporting different block sizes.
- ◆ Derives the **RijndaelManaged** class to provide an implementation of the Rijndael algorithm.

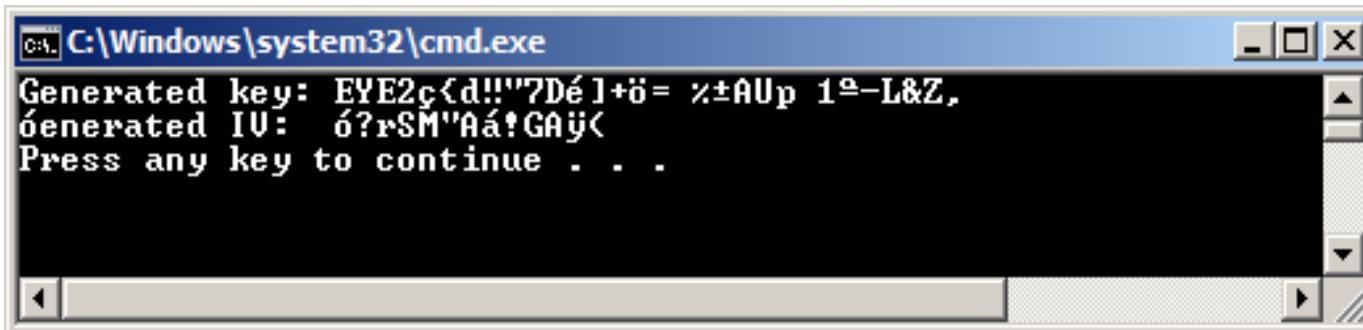
# Performing Symmetric Encryption

- ◆ Developers can use the derived classes of the `SymmetricAlgorithm` base class in the namespace **System.Security.Cryptography** to perform symmetric encryption.
- ◆ This algorithm functions are classified in three steps:
  - ❖ key generation
  - ❖ encryption
  - ❖ decryption

# Generating Symmetric Keys

- When using the default constructor of the symmetric encryption classes, such as **RijndaelManaged** and **AesManaged**, a key and IV are automatically generated.
- The generated key and the IV can be accessed as byte arrays using the Key and IV properties of the encryption class.
- Generating Symmetric Keys

```
RijndaelManaged symAlgo = new RijndaelManaged();
Console.WriteLine("Generated key: {0}, \nGenerated IV: {1}",
Encoding.Default.GetString(symAlgo.Key),
Encoding.Default.GetString(symAlgo.IV));
```



- ◆ The symmetric encryption classes of the .NET Framework provides the CreateEncryptor() method that returns an object of the ICryptoTransform interface.
- ◆ The ICryptoTransform object is responsible for transforming the data based on the algorithm of the encryption class.
- ◆ Once you have obtained an ICryptoTransform object, can use the CryptoStream class to perform encryption.
- ◆ The CryptoStream class acts as a wrapper of a stream-derived class, such as FileStream, MemoryStream, and NetworkStream.

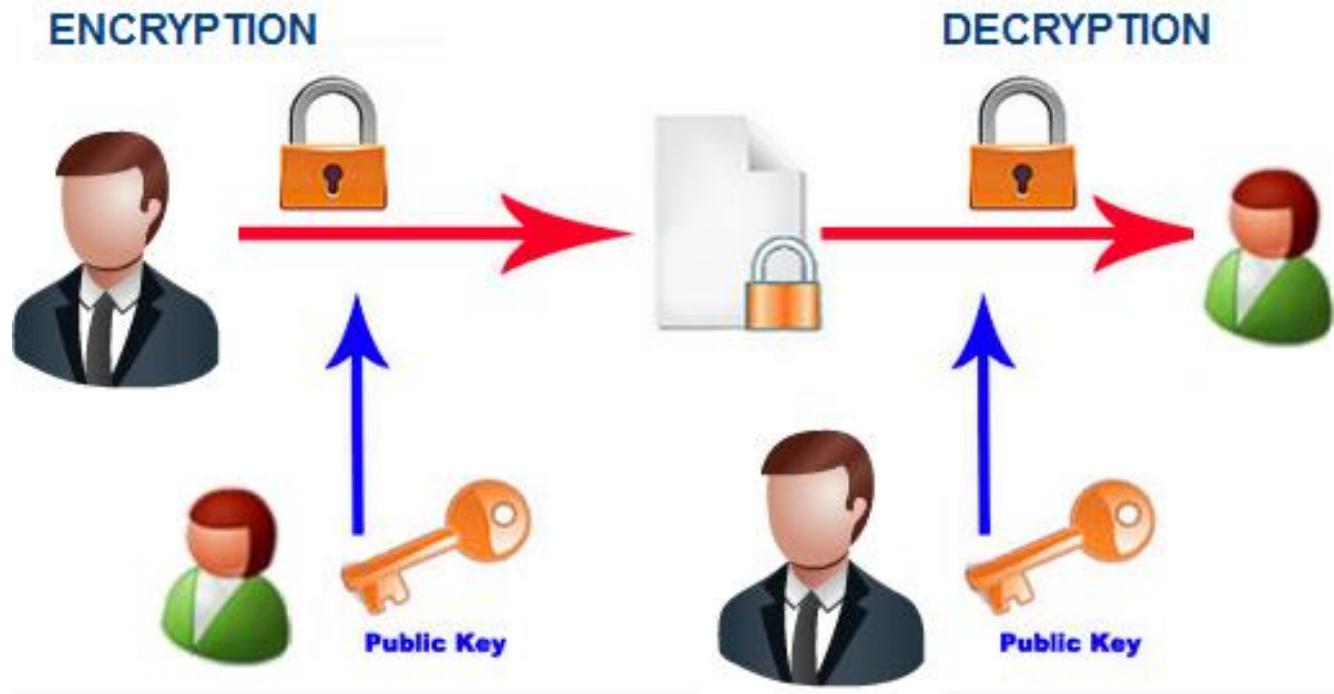
- ◆ To decrypt data, you need to:

- 
- Use the same symmetric encryption class, key, and IV used for encrypting the data.
  - Call the `CreateDecryptor()` method to obtain a `ICryptoTransform` object that will perform the transformation.
  - Create the `CryptoStream` object in Read mode and initialize a `StreamReader` object with the `CryptoStream` object.
  - Call the `ReadToEnd()` method of the `StreamReader` that returns the decrypted text as a string.

- ◆ The **System.Security.Cryptography** namespace provides the **AsymmetricAlgorithm** base class for all asymmetric algorithm classes.
- ◆ **RSA** is an abstract class that derives from the **AsymmetricAlgorithm** class.
- ◆ The **RSA** class is the base class for all the classes that implement the RSA algorithm.
- ◆ The RSA algorithm was designed in 1977 by Ron **Rivest**, Adi **Shamir**, and Leonard **Adleman** and till now is the most widely adopted algorithm to perform asymmetric encryption and decryption.

# Performing Asymmetric Encryption

- You can use the **RSACryptoServiceProvider** class to perform asymmetric encryption.



- ◆ To encrypt data, you need to:
  - ◆ Create a new instance of the **RSACryptoServiceProvider** class
  - ◆ Call the `ImportParameters()` method to initialize the instance with the public key information exported to an `RSAParameters` structure.
- ◆ To decrypt data, you need to initialize an `RSACryptoServiceProvider` object using the private key of the key pair whose public key was used for encryption.



```
RSACryptoServiceProvider rSAKeyGenerator = new  
RSACryptoServiceProvider();  
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);  
RSACryptoServiceProvider rsaDecryptor = new  
RSACryptoServiceProvider();  
rsaDecryptor.ImportParameters(rSAKeyInfo);
```

- ◆ Encryption is a security mechanism that converts data in plain text to cipher text.
- ◆ An encryption key is a piece of information or parameter that determines how a particular encryption mechanism works.
- ◆ The .NET Framework provides various types in the System.Security.Cryptography namespace to support symmetric and asymmetric encryptions.
- ◆ When you use the default constructor to create an object of the symmetric encryption classes, a key and IV are automatically generated.
- ◆ The ICryptoTransform object is responsible for transforming the data based on the algorithm of the encryption class.
- ◆ The CryptoStream class acts as a wrapper of a stream-derived class, such as FileStream, MemoryStream, and NetworkStream.
- ◆ When you call the default constructor of the RSACryptoServiceProvider and DSACryptoServiceProvider classes, a new public/private key pair is automatically generated.

Session: 17

# More Features of C# 7.0 and 7.1

- ◆ Describe `ref` returns and `ref` locals
- ◆ Explain improvements made to `out` variables, tuples, asynchronous `Main()`, and `throw` expressions
- ◆ Identify the new expression-bodied members

## Key new features introduced in C# 7.0 and 7.1

- Choosing the preferred language version
- Ref returns, ref locals, and improved out variables
- Improved tuples
- Improved asynchronous Main()
- Improved throw expressions
- More expression-bodied members



C# 7.0      C# 7.1

# **ref Returns, ref Locals, and Improved out Variables**

In C#, a method can contain a parameter that a developer can pass by reference using the `ref` keyword.

It is compulsory to specify the `ref` keyword in the calling method as well as in the definition of the called method.



- Following is a sample code snippet which demonstrates how the `ref` keyword stores values passed by reference in local variables:

### Snippet

```
class Program
{
    static void Main(string[] args)
    {
        string[] writers = {"Emy George", "Lee Mein", "John Wash",
        "Sicily Wang"};
        ref string writer2 = ref new Program().FindWriter(1, writers);
        Console.WriteLine("Original writer:{0}", writer2); Console.WriteLine();
        writer2 = "Johan Muller";
        Console.WriteLine("Replaced writer:{0}", writers[1]); Console.ReadKey();
    }
    public ref string FindWriter(int num, string[] names)
    {
        if (names.Length > 0)
            return ref names[num];
        throw new IndexOutOfRangeException($"nameof({num}) unavailable.");
    }
}
```

- ◆ Following code snippet demonstrates that it is possible to save a reference in a local variable:

```
class Program {
    public static object ReturnChars { get; private set; }
    static void Main(string[] args)
    {
        Console.WriteLine("Input a string");
        char[] cseq = Console.ReadLine().ToCharArray();
        Console.WriteLine($"Prior to replacing: { new string(cseq) }");
        ref char cref = ref RetRefLocal.SeekCharRef(cseq[0],cseq);
        cref = 'p';
        Console.WriteLine($"Post replacing: {new string (cseq)}");
        Console.ReadLine();
    }
    class RetRefLocal {
        public static ref char SeekCharRef(char val, char[] cSeq)
        {
            for (int k = 0; k < cSeq.Length; k++)
            {
                if (cSeq[k] == val)
                {
                    return ref cSeq[k];
                }
            }
            throw new IndexOutOfRangeException(val + "not there");
        }
    }
}
```

# Improved out Variables and Discards

- ◆ Following code snippet demonstrates how developers can specify the data type of all the `out` parameters inline:

## Snippet

```
class BookApplication
{
    static void Main(string[] args)
    {
        BookByOutArg(out string bName, out string bAuthor);
        Console.WriteLine("Book: {0}, Author: {1}", bName, bAuthor);
        Console.ReadKey();
    }

    static void BookByOutArg(out string name, out string author)
    {
        name = "Harry Potter Part I ";
        author = "J. K. Rowling";
    }
}
```

# Improved Tuples (1-3)

With C# 7.1, a new feature called as tuple name inference has been added.



## Tuples

Tuple name inference also enables tuples to deduce the names of their elements from the inputs.

Using tuple name inference, one can work with tuples as value types.

- Following code snippet shows how two elements of a tuple have been defined without any name inference:

### Snippet

```
class Program  {
    static void Main()
    {
        string ename="Emy George";
        int e_age = 30;
        var empTuple = (ename, e_age);
        Console.WriteLine(empTuple.Item1); //Emy George
        Console.WriteLine(empTuple.Item2); //30
        Console.ReadKey();
    }
}
```

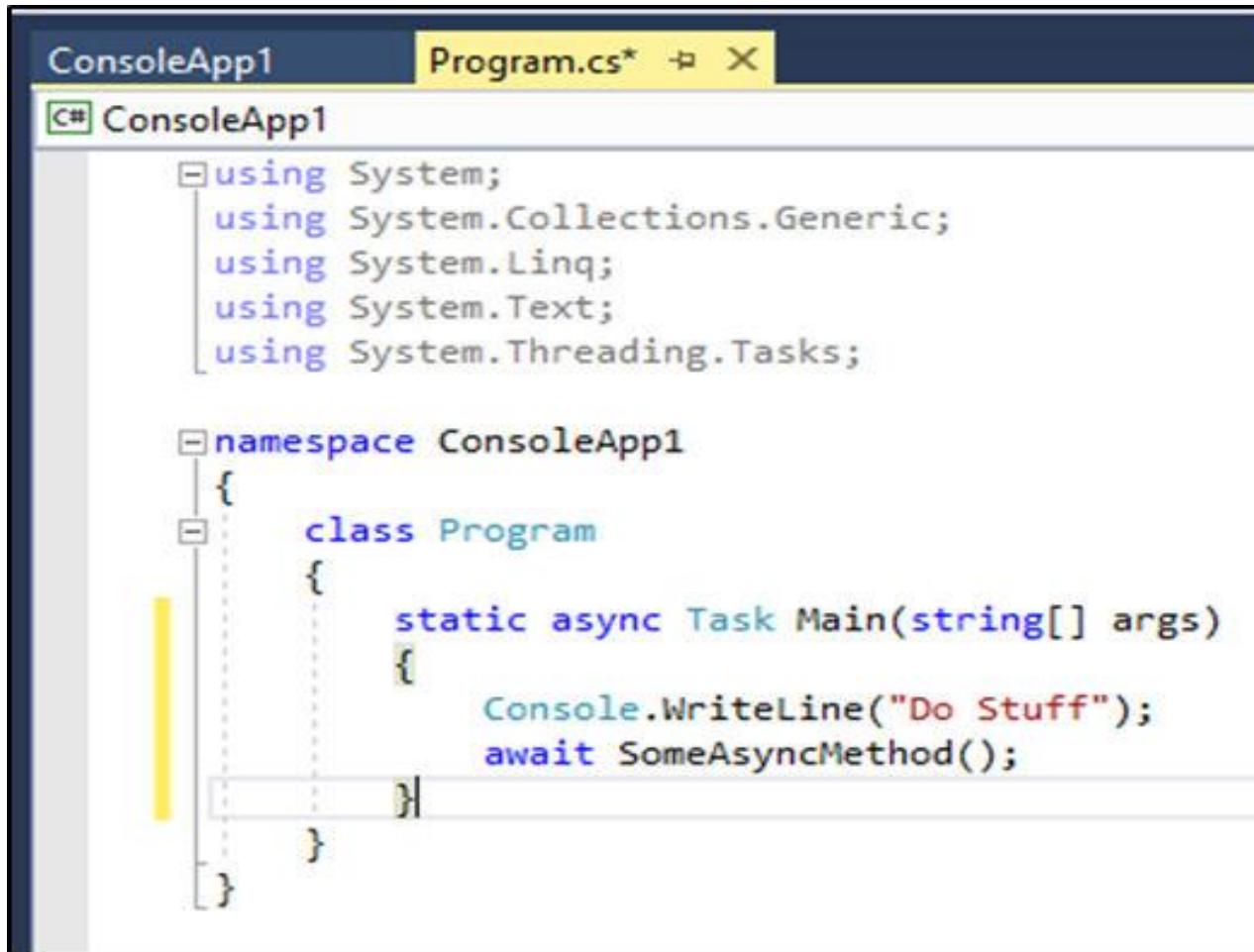
- Following code snippet demonstrates that how to specify element names manually:

### Snippet

```
string ename = "Emy George";
int e_age = 30;
var empTuple = (ename: ename, e_age: e_age);
Console.WriteLine(empTuple.ename); //Emy George
Console.WriteLine(empTuple.e_age); //30
```

## Improved Asynchronized Main()

- Following figure shows how it is easily possible to specify the Main() method of a C# 7.1 application as async:



The screenshot shows a code editor window titled "ConsoleApp1" with the file "Program.cs\*" selected. The code is written in C# and defines an asynchronous Main method:

```
ConsoleApp1
C# ConsoleApp1

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Console.WriteLine("Do Stuff");
            await SomeAsyncMethod();
        }
    }
}
```



# Improved Throw Expressions

- Following code snippet demonstrates how the `throw` statement was used previously as a separate statement:

## Snippet

```
if (num1 == null) {
    throw new ArgumentNullException(nameof(num1));
}
```

- Following code shows how C# 7.0 uses `throw` expression write code in a more concise manner.

## Snippet

```
myNum = num1 ?? throw new ArgumentNullException(nameof(num1));
```

- Following code demonstrates how to use the `?:` operator to represent an if/else statement:

## Snippet

```
return val < 10 ? val : throw new
ArgumentOutOfRangeException("Value has to less
than 10");
```

## More Expression-bodied Members

With C# 7.0, developers can now extend the expression-bodied members feature to cover more members such as property accessors, destructors, and constructors.

- ◆ Following code snippet demonstrates how an expression-bodied method can be used to invoke an asynchronous method in `Main()`:

### Snippet

```
static async Task Main(string[] args) =>  
    WriteLine($"Factorial 6: {await AsFact(6)}");
```

C#

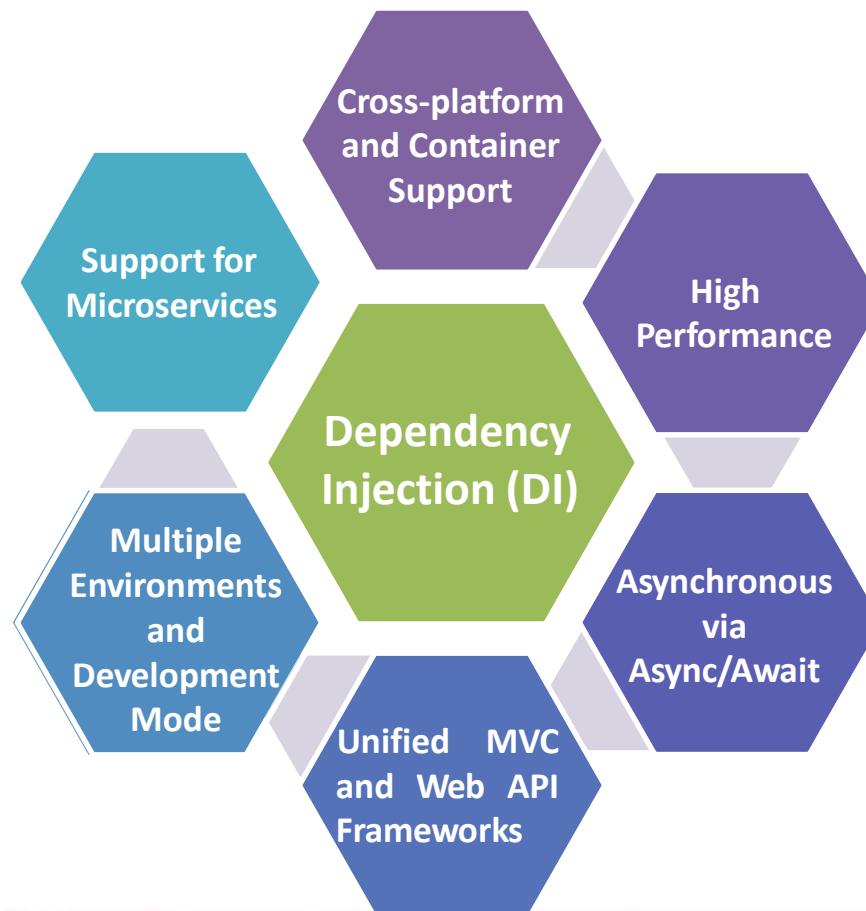
- ◆ The `ref` keyword allows returning and storing values passed by reference.
- ◆ The `ref returns` and `ref locals` features are useful for replacing placeholders or reading from large data structures.
- ◆ C# 7.1 allows specifying the data type of out parameters inline in a method.
- ◆ The compiler for C# 7.1 is capable of inferring the element names of a tuple from local variables, null conditional members, and other members such as properties.
- ◆ C# 7.1 allows adding a throw exception in null-coalescing and conditional expressions and expression-bodied members.
- ◆ Expression-bodied members can include not only methods but also constructors, destructors, and properties, and property accessors.

Session: **18**

# .NET Core Development

- ◆ Explain what is .NET Core
- ◆ List the differences between .NET Core and .NET Framework
- ◆ Explain how to run a program on .NET Core

- .NET Core is an open source .NET Framework platform available for Windows, Linux, and Mac OS.
- Following are the key features of .NET Core:



Where

- Docker containers are being employed.
- Server apps can be set up cross-platform to Docker containers.
- For containers.

When

- There is a need for a high performance and scalable system.
- Users are executing several .NET versions side-by-side.
- To set up applications that have dependencies on diverse versions of frameworks.
- Command Line Interface (CLI) control is required.

- Following are some of the features that are currently not supported in .NET Core and hence, not recommended in situations that need these:

Certain.NET  
features and  
libraries and  
extensions

Windows Forms  
and WPF  
applications

Third-party  
library support

ASP.NET Web  
Forms

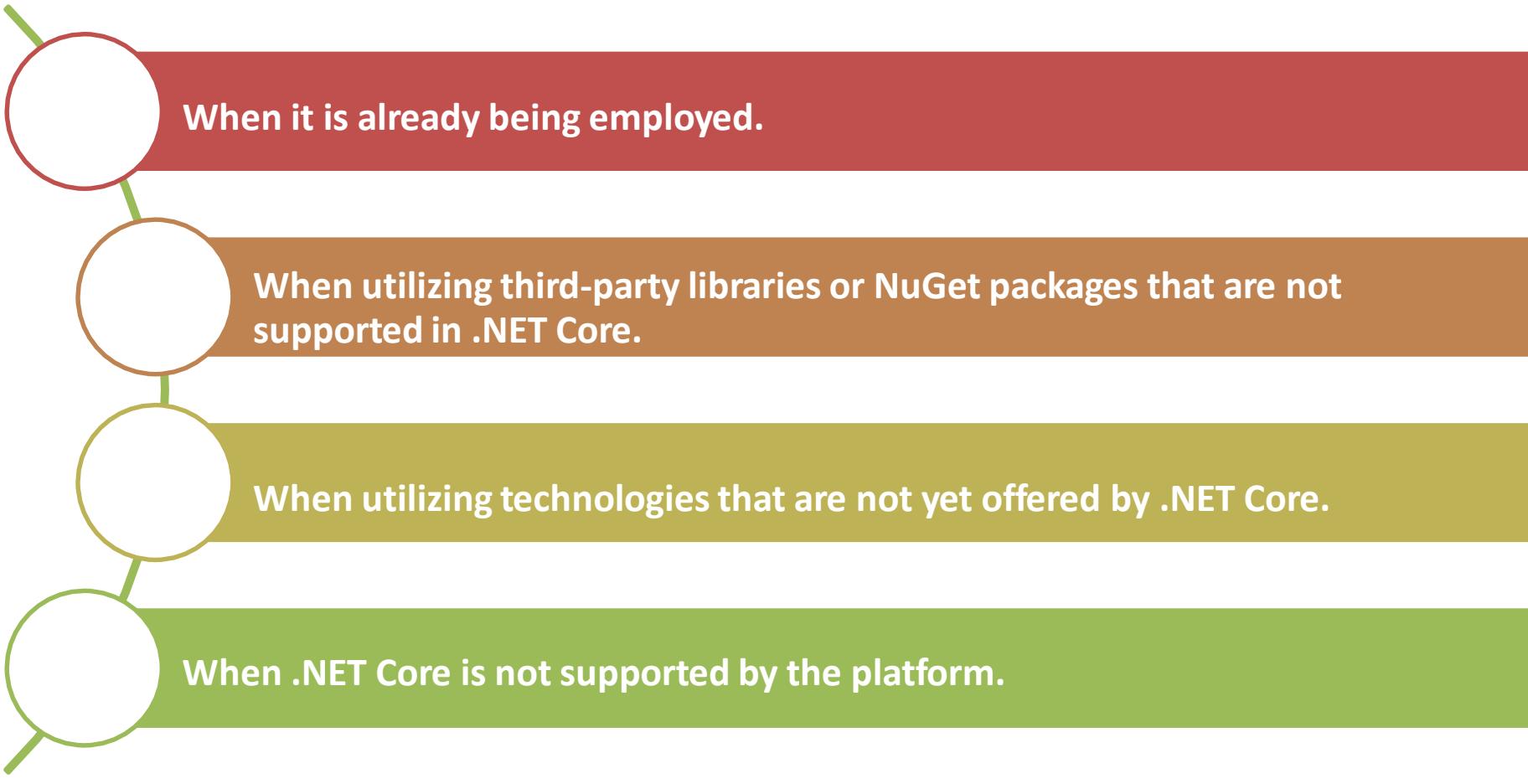
Partial support  
for VB.NET and F#

Windows  
Communication  
Foundation  
(WCF)

SignalR

# .NET Framework in Comparison with .NET Core(1-2)

As .NET Framework can also work with Docker and Windows Containers, utilizing it in the following situations is possible:



When it is already being employed.

When utilizing third-party libraries or NuGet packages that are not supported in .NET Core.

When utilizing technologies that are not yet offered by .NET Core.

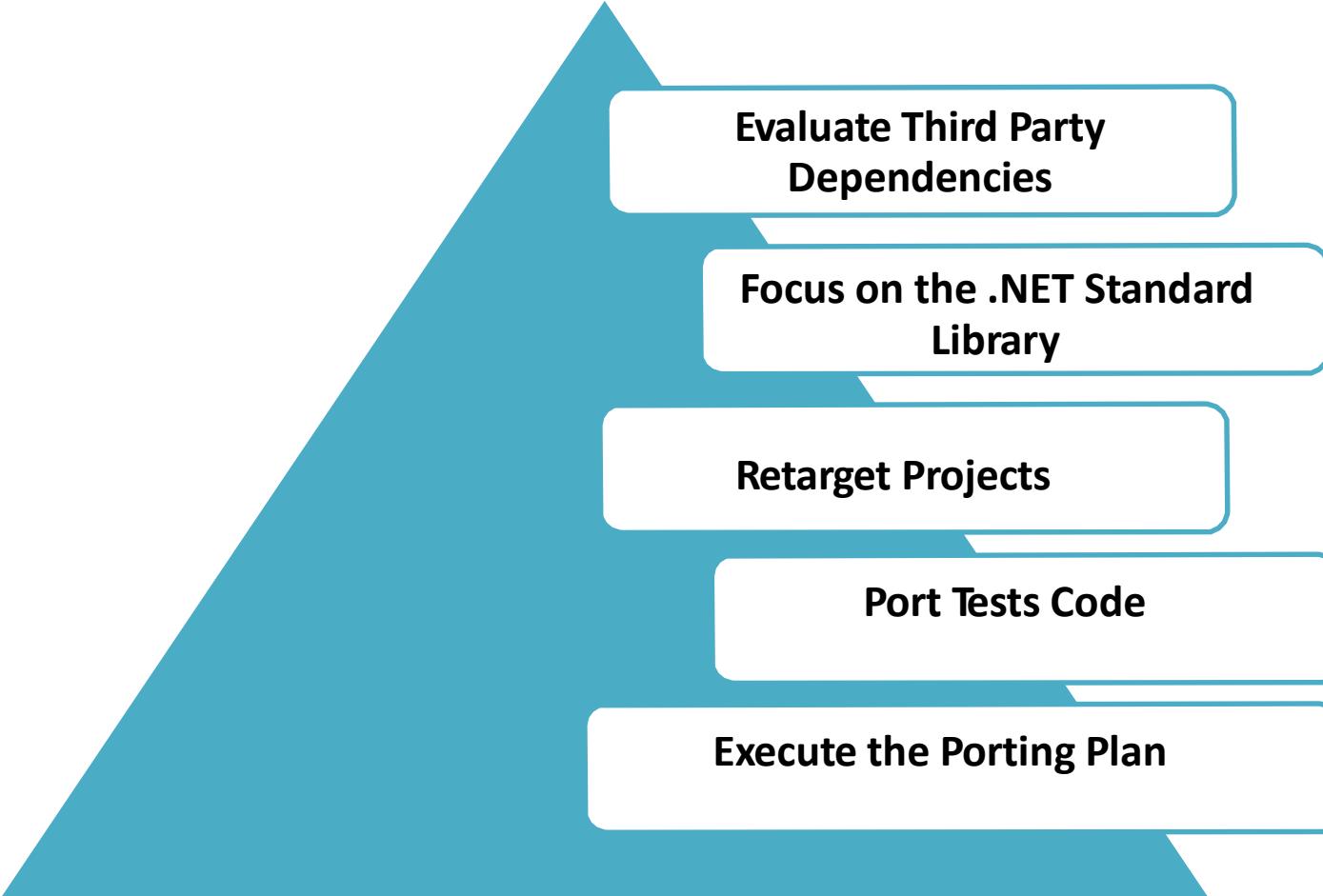
When .NET Core is not supported by the platform.

Following are the situations in which .NET Framework should not be used:

- ◆ When several OS platforms are needed.
- ◆ When high performance and scalability are required.
- ◆ When .NET Core works.
- ◆ When open source framework is needed.



# Porting from .NET Framework to .NET Core



Evaluate Third Party Dependencies

Focus on the .NET Standard Library

Retarget Projects

Port Tests Code

Execute the Porting Plan

# Differences Between .NET Framework and .NET Core (1-3)

.NET Framework	.NET Core
<b>It was made available as a licensed and proprietary software framework.</b>	<b>It was released as an open source software framework.</b>
<b>It allows users to create applications for a single platform - Windows.</b>	<b>It accommodates three different OSes - Windows, OS X, and Linux.</b>
<b>It must be set up as a single package and runtime environment for Windows.</b>	<b>It can be packaged and set up irrespective of the underlying OS.</b>
<b>It uses CLR for managing libraries and compilation purposes.</b>	<b>It utilizes a redesigned CLR known as CoreCLR and a modular collection of libraries known as CoreFX.</b>
<b>If utilizing Web applications, users can employ a robust Web application framework, such as ASP.NET.</b>	<b>It has a redesigned version of ASP.NET.</b>

## Differences between .NET Framework and .NET Core (2-3)

.NET Framework	.NET Core
<b>Users must deploy Web applications only on Internet Information Server.</b>	<b>The Web applications can be run in several ways.</b>
<b>It has extensive APIs for creating cloud-based applications.</b>	<b>It has features that ease the creation and deployment of Cloud-based application.</b>
<b>It does not have any robust framework or tools to simplify mobile app development.</b>	<b>It is compatible with Xamarin via the .NET Standard Library.</b>
<b>It provides few options for developing microservice oriented systems</b>	<b>It allows the creation of microservice oriented systems rapidly.</b>
<b>It requires additional infrastructure to achieve scalability.</b>	<b>It enhances the performance and scalability without having to deploy extra hardware or infrastructure.</b>

## Differences between .NET Framework and .NET Core (3-3)

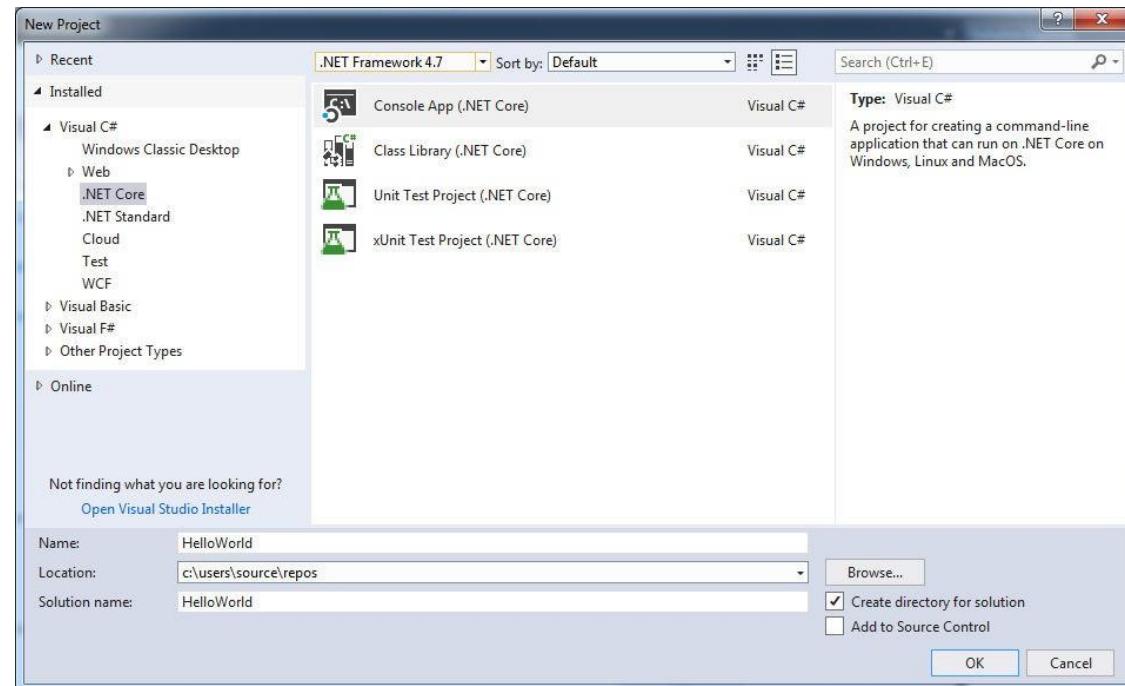
Following table summarizes which of the two (.NET Framework or .NET Core) is preferred:

Feature	.NET Framework	.NET Core
High-performance and scalable system without UI		✓
Docker containers support	✓	✓
Heavily rely on command line		✓
Cross-platform needs		✓
Using microservices	✓	✓
UI centric Web applications	✓	
Windows client applications using Windows Forms and WPF	✓	
Already have a pre-configured environment and systems	✓	
Stable version for immediate need to build and deploy	✓	
Has experienced .NET team		✓
Time is not an issue, experiments are acceptable and no rush to deployment		✓

# Running a Program on .NET Core (1-2)

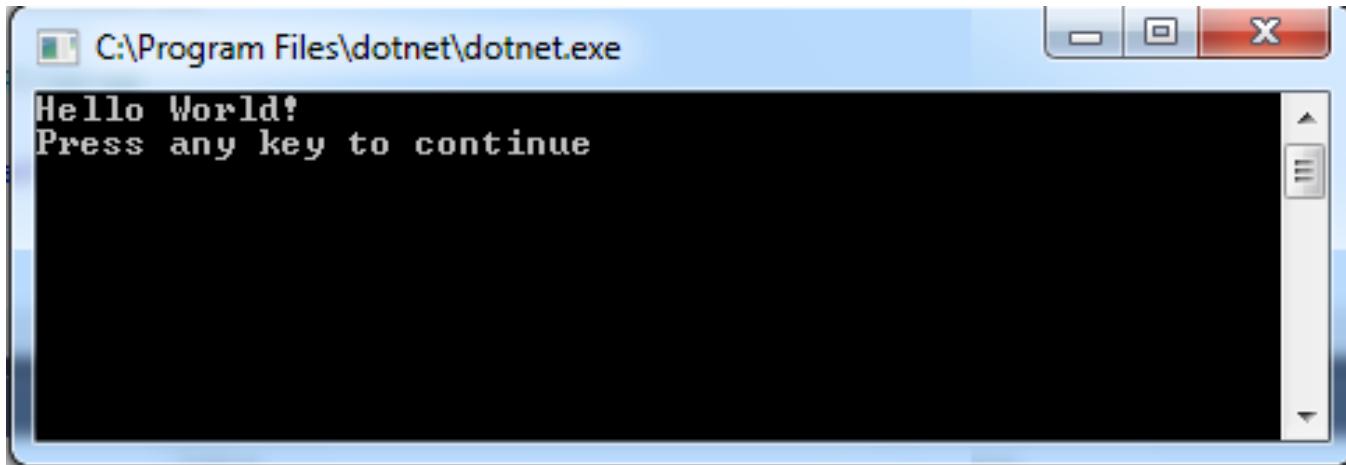
Following are the steps to create a simple ‘Hello World’ console application:

1. Start Visual Studio 2017.  
Click File→New→Project.  
The New Project dialog appears as shown in the figure.
  - a. Choose Visual C# followed by the .NET Core node.
  - b. Next, choose the Console App (.NET Core) project template.
  - c. Provide the name 'HelloWorld' in the Name field and click OK.



New Project Dialog Box

2. On the menu bar, click Build → Build Solution.
3. Execute the program. Following figure shows the output of the program.



Output of the Program

- ◆ .NET Core is an open source .NET Framework platform available for Windows, Linux, and Mac OS.
- ◆ The key features of .NET Core are Cross-platform and container support, High performance, Asynchronous via `async/await`, Unified MVC and Web API Frameworks, Multiple environments and development mode, Dependency injection and Support for Microservices.
- ◆ .NET Core is a perfect fit if there is a need for a high performance and scalable system.
- ◆ Microsoft suggests executing .NET Core along with ASP.NET Core to obtain best performance and scale.
- ◆ .NET Core is usable if users are executing several .NET versions side-by-side.
- ◆ Though there are a lot of instances when .NET Core can be utilized, there are situations in which it is not recommended for use.

Session: **19**

# Debugging C# in Visual Studio 2017

- ◆ Describe the concept of debugging in Visual Studio 2017
- ◆ Identify the different ways of debugging C# applications
- ◆ Explain the process of installing, updating, and removing NuGet packages

Debugging is a process that tracks an application by going through its each line of code.

## Visual Studio 2017

Allows debugging of C# applications.

Indicates status of running code.

Provides wide set of debugging tools.

Allows changing values to see the possible outcome while the application is in its running state.

## Default Build Configurations

**Release**

- Release configuration is for distributing the final version.

**Debug**

- Debug configuration is for debugging the application code.

Figure indicates that Visual Studio is set to compile the opened application in the Debug mode.



Debug Mode

# Debugging Tools in Visual Studio 2017 2-2

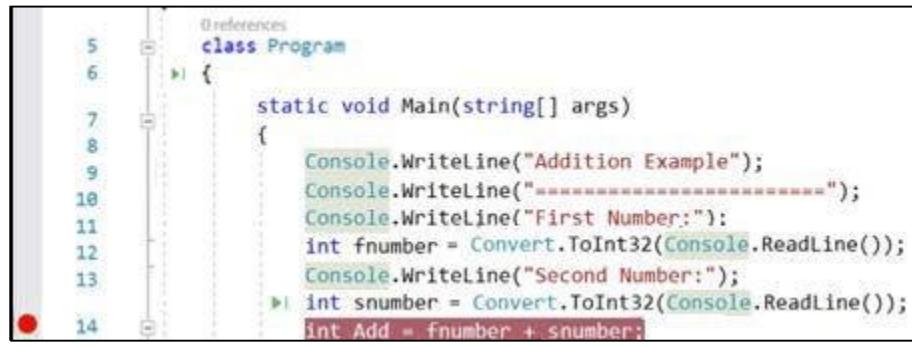
The screenshot shows the Visual Studio 2017 IDE interface during a debugging session. The main window displays the code for a C# console application named 'ConsoleAppDemo'. The code defines a 'Program' class with a 'Main' method that reads two numbers from the console, adds them, and prints the result. A break point is set on the line 'int Add = fnumber + snumber;'. The 'Diagnostic Tools' window is open, showing memory usage and CPU usage over time. The 'Call Stack' window shows the current call stack entry: 'ConsoleAppDemo.dll!ConsoleAppDemo.Program.Main(string[] args) Line 9'. The bottom navigation bar includes tabs for 'C# Interactive', 'Locals', and 'Watch 1'.

```
using static System.Console;
namespace ConsoleAppDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Addition Example");
            Console.WriteLine("-----");
            Console.WriteLine("First Number:");
            int fnumber = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Second Number:");
            int snumber = Convert.ToInt32(Console.ReadLine());
            int Add = fnumber + snumber;
            Console.WriteLine("Addition is:" + Add);
            Console.ReadLine();
        }
    }
}
```

## Debugging Environment in Visual Studio 2017

## Breakpoint:

- ◆ Refers to a signal that stops the application's execution at the already set points in the editor.
- ◆ Pauses execution prior to running the line of code having the breakpoint.
- ◆ Waits for the next instruction or command to proceed.
- ◆ Allows breaking the execution at any line of code with the help of debugger.



The screenshot shows a code editor window with a C# program named 'Program'. The code defines a static void Main method that reads two numbers from the console and adds them together. A red circular breakpoint marker is placed on the left margin at line 14, which contains the line 'int Add = fnumber + snumber;'. The code is syntax-highlighted, with 'Console' in green and 'WriteLine' in blue.

```
0 references
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Addition Example");
        Console.WriteLine("-----");
        Console.WriteLine("First Number:");
        int fnumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Second Number:");
        int snumber = Convert.ToInt32(Console.ReadLine());
        int Add = fnumber + snumber;
    }
}
```

Breakpoint

### Ways to run debugger:

Press F5

Click green-play Icon on the toolbar

Select Debug → Start Debugging



## Data Tip:

- ◆ Informative tooltips available only in break mode.
- ◆ Reveals more information about a selected variable or an object in the present execution scope.
- ◆ Can be pinned and can be commented.

The screenshot shows a code editor window with the following C# code:

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Addition Example");
        Console.WriteLine("=====");
        Console.WriteLine("First Number:");
        int fnumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Second Number:");
        int snumber = Convert.ToInt32(Console.ReadLine());
        int Add = fnumber + snumber;
        Console.WriteLine("Addition is:" + Add);
        Console.ReadLine();
    }
}
```

A tooltip is displayed for the variable `fnumber`, which contains the value `2`. The tooltip has a comment input field below it with the placeholder `Type a comment here`.

***Data Tip***

# Using Windows and Conditional Breakpoints 1-3

If a developer sets a breakpoint on the line that prints user input, the Autos, Locals, Immediate and Watch windows are quite useful to observe.

## Autos

A window that displays the information of all variables and objects such as type and value that are in the present or the previous line.

Autos		
Name	Value	Type
Add	0	int
finumber	2	int
snumber	3	int

Autos Window

## Locals

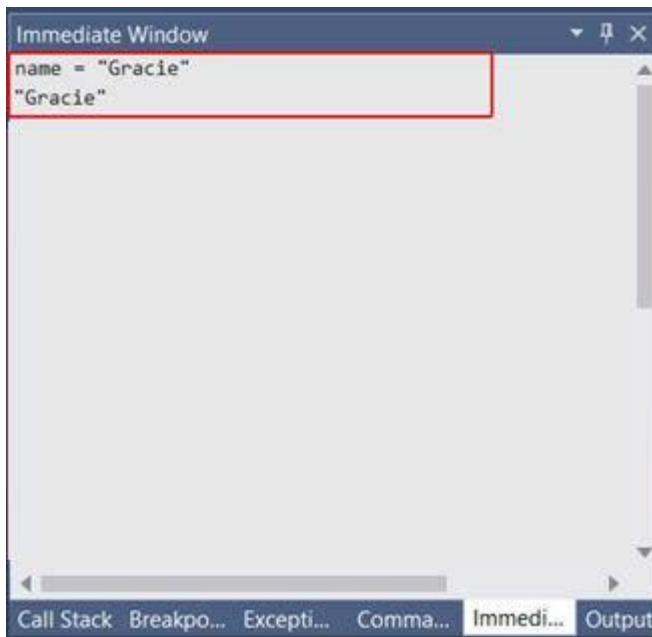
Shows the values of local variables and objects that are within the scope of current execution.

Locals		
Name	Value	Type
string.Concat returned	"Addition is5"	string
args	[string[0]]	string[]
finumber	2	int
snumber	3	int
Add	5	int

Locals Window

## Using Windows and Conditional Breakpoints 2-3

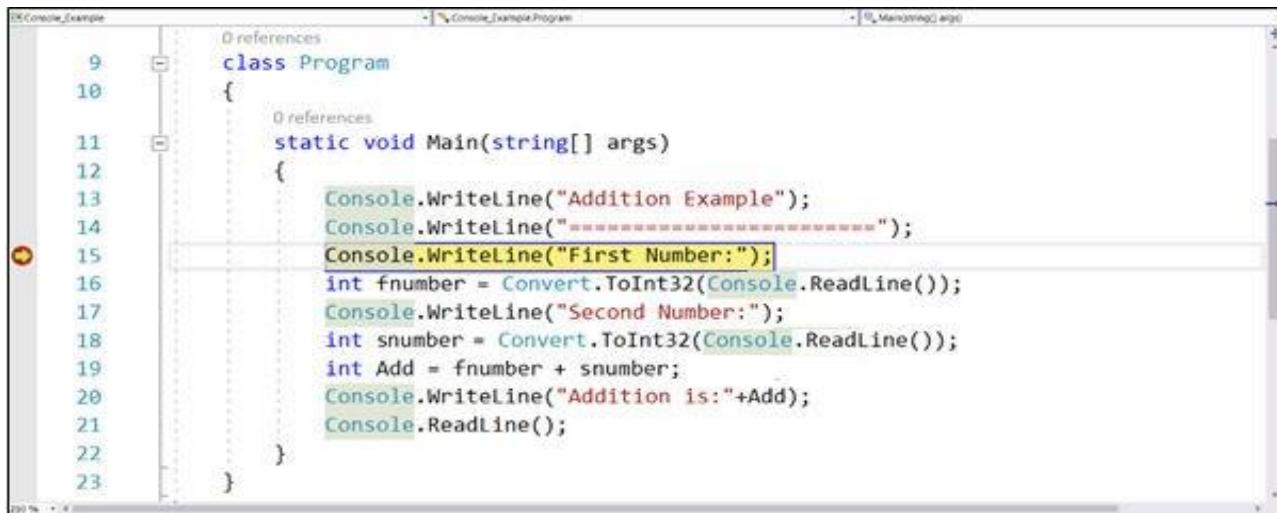
**Immediate:** A window that allows changing the values.



**Immediate Window**

**Watch:** A window that display information about variables and objects depending on the changes made to them.

# Using Windows and Conditional Breakpoints 3-3

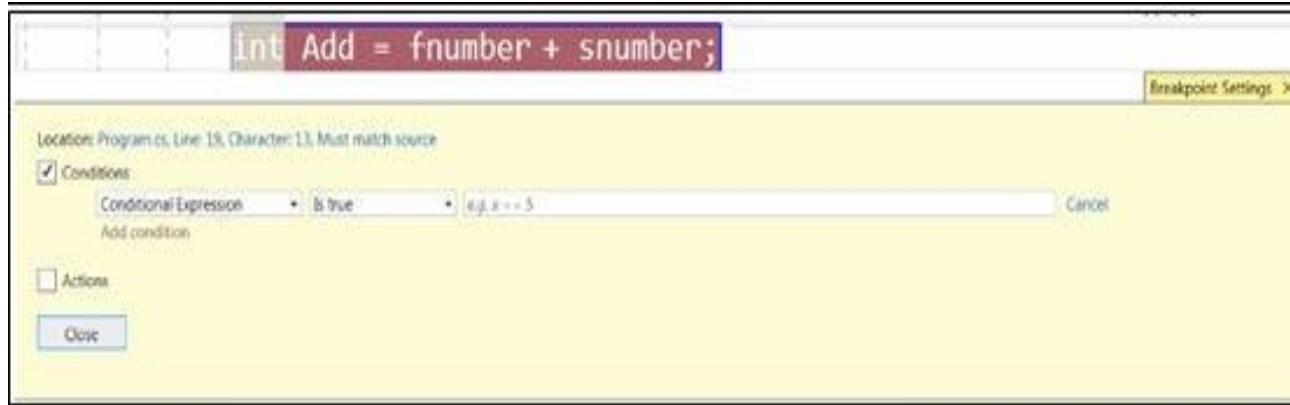


The screenshot shows the Microsoft Visual Studio IDE. In the center is a code editor window titled 'Console\_Example.cs' containing the following C# code:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Addition Example");
        Console.WriteLine("-----");
        Console.WriteLine("First Number:");
        int fnumber = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Second Number:");
        int snumber = Convert.ToInt32(Console.ReadLine());
        int Add = fnumber + snumber;
        Console.WriteLine("Addition is:" + Add);
        Console.ReadLine();
    }
}
```

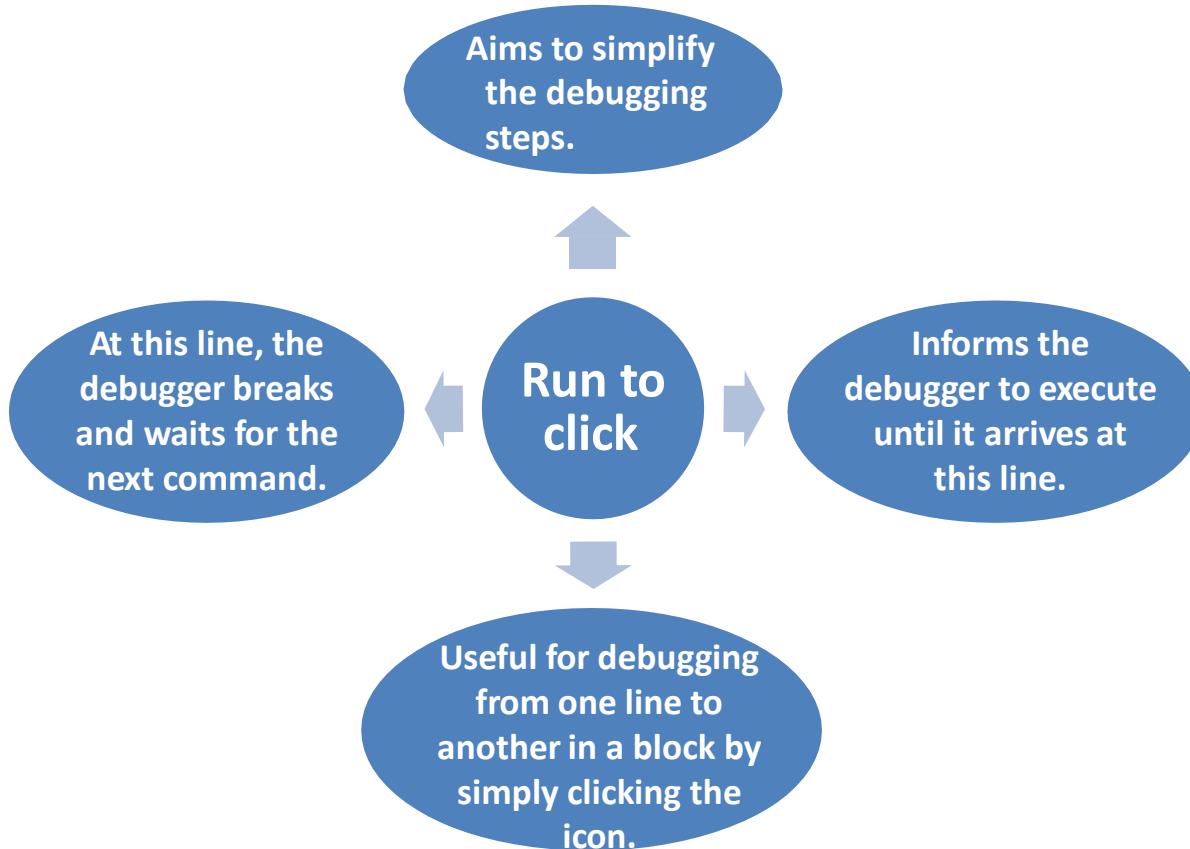
A red circle highlights the breakpoint at line 15, which is the first line of the Main method body. The status bar at the bottom left indicates '200 %'.

Code in Debug Mode



Breakpoint Settings Dialogue Box

# Using Run to Click



A screenshot of a debugger interface showing the following code:

```
18 int snumber = Convert.ToInt32(Console.ReadLine());
19
20 int Add = fnumber + $number;
21 Console.WriteLine("Addition is:" +Add);
Console.ReadLine();
```

The cursor is positioned at line 20, and a tooltip indicates "Run to click here".

## Run to Click Feature

## NuGet and Advantages

A free extension for Visual Studio that makes it efficient to look for all the open-source packages and use them in projects.

Makes it simple to add, update, and delete libraries in projects that rely on the .NET Framework.

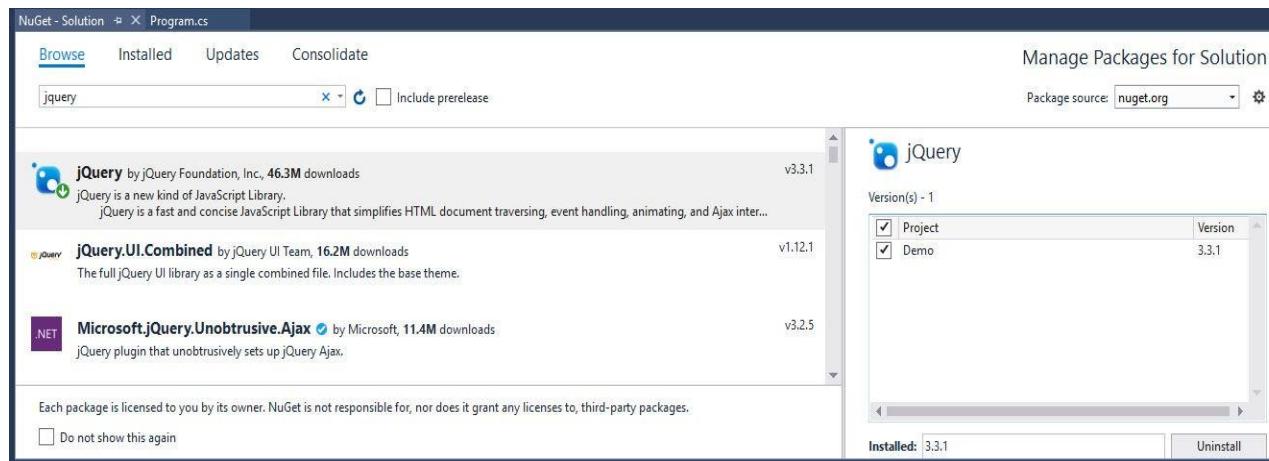
Is a free package management system, which allows sharing code.

Allow developers to reuse their own code across their own projects.

Automates the tasks of installing, setting up, and updating packages or components in a Visual Studio project.

# Using NuGet to Install a Package 1-3

1. Go to **Solution Explorer**.
2. Right-click a project solution.
3. Select **Manage NuGet Packages for Solution**. The NuGet – Solution window is displayed.  
This is the NuGet Package Manager.



**NuGet – Solution Window**

4. Enter the name of the desired package in the search field on the top.
5. Select the package from the list box and its corresponding details such as description, version, author, and license.
6. Select a suitable version from the Version drop-down list.

# Using NuGet to Install a Package 2-3

Swashbuckle.AspNetCore by Swashbuckle.AspNetCore, 170K downloads  
Swagger tools for documenting APIs built on ASP.NET Core v1.0.0

Threax.AspNetCore.Swashbuckle.Convention by Threax.AspNetCore.Swashbuckle.Convention, 4K downloads  
Threax.AspNetCore.Swashbuckle.Convention v1.5.0

Swashbuckle.AspNetCore.Examples by Swashbuckle.AspNetCore.Examples, 155 downloads  
Package Description v1.2.0

Swashbuckle.Core.Net45 by Richard Morris, 98.5K downloads  
Combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery, documentation and playground experience to your API consumers v5.2.1

Swashbuckle.Net45 by Richard Morris, 49.7K downloads  
Combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery, documentation and playground experience to your API consumers v5.2.1

**Swashbuckle.Net45**

Version: Latest stable 5.2.1

Options

Description  
Seamlessly adds a Swagger to WebApi projects! Compiled for .NET 4.5 and up. This helps with mono compatibility and does not require any assembly redirects.

Version: 5.2.1

Author(s): Richard Morris

License: <http://opensource.org/licenses/BSD-3-Clause>

Date published: Wednesday, August 12, 2015 (8/12/2015)

Project URL: <https://github.com/SEEK-Jobs/Swashbuckle>

## Package Details

7. Click **Install** to add the package.
8. Accept all prompts regarding licensing. The **Choose NuGet Package Manager Format** dialog box will be displayed.
9. Click **OK**. The Output window at the bottom of Visual Studio IDE displays a message that the package is installed successfully.
10. For installing a package, it is essential to specify the right source for it.



## Package Source

Following are some ways to install a NuGet package:

By using command window in the project folder and **dotnet.exe** command as: `dotnet add package <package_name>` which obtains the package as per its name, puts its contents into the current directory, and refers to it

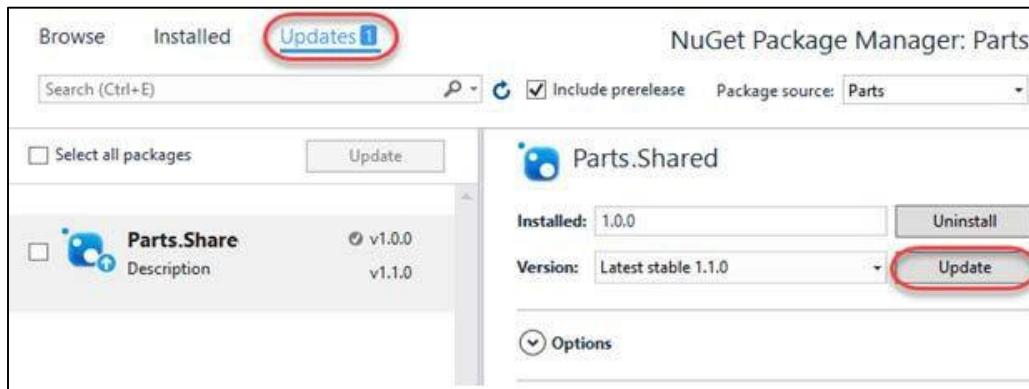
By using the Package Manager Console (**Tools → NuGet Package Manager → Package Manager Console**) and entering the install package `<package_name>` command. The command obtains the package and its dependencies from a chosen source and installs it into the project

By using **nuget.exe** and entering the nuget package `<package_name>` command



# Using NuGet to Update a Package

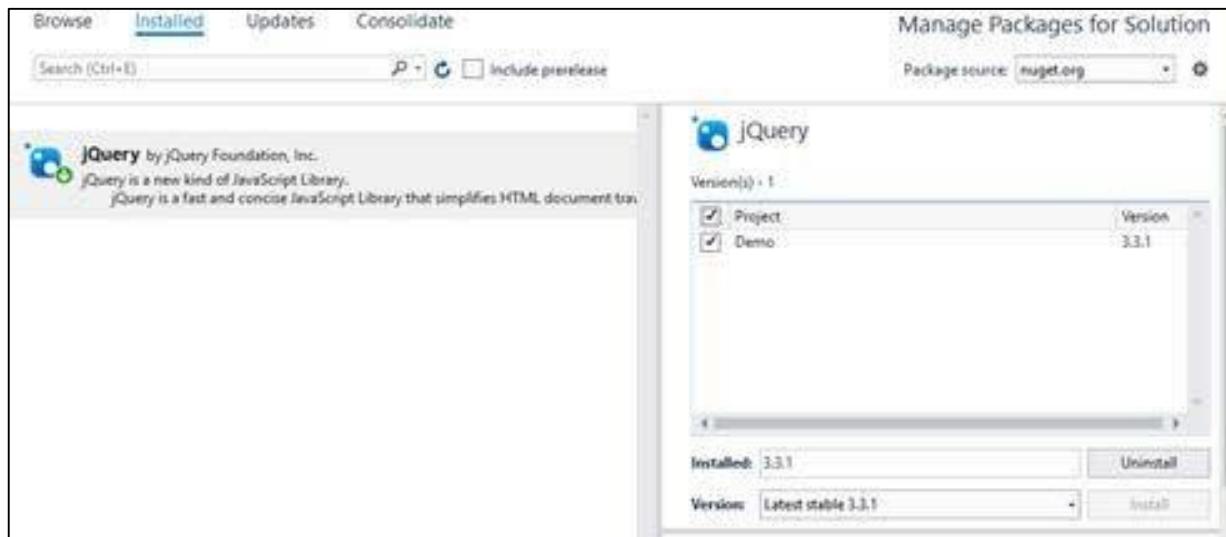
The **Updates** tab of the **NuGet – Solution** window allows updating the installed packages. It shows the packages whose updates are available from the chosen package sources.



**Updates Tab**

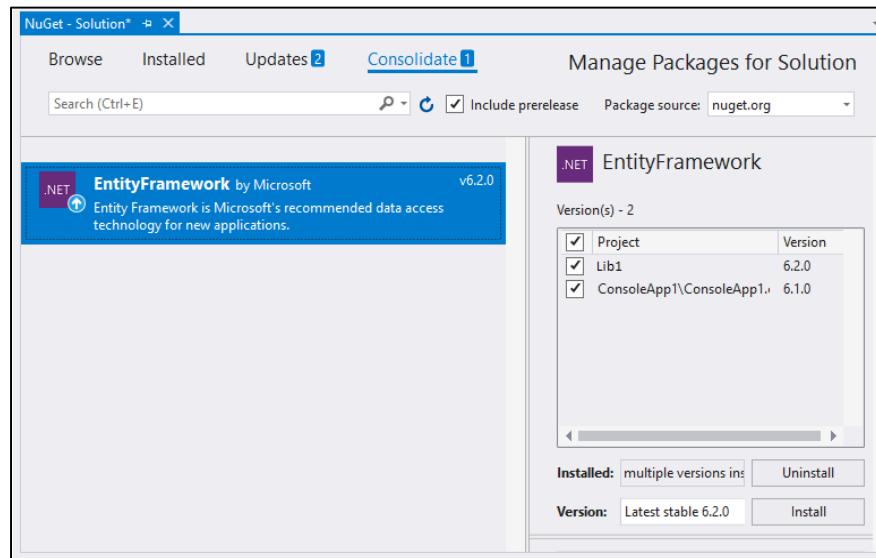
# Using NuGet to Uninstall a Package

The **Installed** tab of the **NuGet – Solution** window allows updating or uninstalling the installed packages.



**Installed Tab**

If different versions of NuGet package are in use, it is possible to manage or merge them using the **Consolidate** tab of the **NuGet – Solution** window.



**Consolidate Tab**

- ◆ Developers should start testing an application in the debug mode, as it provides comprehensive details while building the application.
- ◆ The Visual Studio debugger allows inspecting what the targeted program does while it is executing.
- ◆ In the debug mode, breakpoints as special indicators inform the debugger to stop the program's execution when it arrives at a targeted line of code.
- ◆ Visual Studio allows executing the current line and then move to the next one using the **Debug → Step Over** option.
- ◆ The **Debug → Step Into** option allows executing each line of code within a method or functionality for troubleshooting it.
- ◆ The **Autos**, **Locals**, and **Immediate** windows allow viewing the status and values of variables and objects while debugging.

- ◆ The Run to Click feature is useful for debugging from one line to another in a block by simply clicking the icon rather than inserting breakpoints.
- ◆ NuGet is a free extension in Visual Studio 2017 that allows integrating the third-party packages in one to two clicks in a project.
- ◆ A NuGet package is a ZIP file containing all compiled code, a manifest file, and some more files associated with the code.
- ◆ In Visual Studio 2017, the integrated NuGet Package Manager installs, updates, and removes packages.