AGENDA

- Variables, Data Types, Flow Control, Enumeration, Arrays, Namespaces, Main() Method, Compiling C# File, Console I/O, Comments, Rules for Identifiers, Class Members
- **Inheritance:**
- Types, Implementation, Abstract Class, Sealed Class, Modifiers, Interfaces
- Operators, Type Safety, Comparing Objects for Equality, User-Define Casts
- **Advanced C#:**
- Memory Management, Feeing Unmanaged Resources, Unsafe Code, String Class, Error and Exception Handling, Delegates and Events.

Introduction

- C# is a modern, general-purpose object oriented programming language developed by Microsoft and approved by Ecma and ISO.
- C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.
- C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages to be used on different computer platforms and architectures.

C# - Overview

- **The following reasons make C# a widely used professional language:**
- Modern, general purpose programming language
- Object oriented.
- Component oriented.
- Easy to learn.
- Structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- Part of .Net Framework.

C# - Overview

- Although C# constructs closely follows traditional high level languages C and C++ and being an object oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

C# - Overview

Strong Programming Features of C#

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy to use Generics
- Indexers
- Conditional Compilation
- Simple Multithreading
- LINQ and Lambda Expressions
- Integration with Windows

C# - Environment

The .Net framework consists of an enormous library of codes used by the client languages like C#:

- Common Language Runtime (CLR)
- The .Net Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.Net and ASP.Net AJAX
- ADO.Net
- Windows Workflow Foundation (WF)
- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- LINQ

C# - Environment

Integrated Development Environment (IDE) For C#

- Microsoft provides the following development tools for C# programming:
- Visual Studio 2019 (VS) and others.
- Visual C# 2019 Express (VCE)
- Visual Web Developer
- Visual C# Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same look and feel. They retain most features of Visual Studio.

C# - Environment

- **Writing C# Programs on Linux or Mac OS**
- Although the .NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems.
- **Mono** is an open-source version of the .NET Framework, which includes a C# compiler and runs on several operating systems, including various flavors of Linux and Mac OS.
- The stated purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools to Linux developers.
- Mono can be run on many operating systems including Android, BSD, iOS, Linux, OS X, Windows, Solaris and UNIX.

C# - Program Structure

A C# program basically consists of the following parts:

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements & Expressions
- Comments

C# - Program Structure

```
using System;
namespace HelloWorldApplication
{
  class HelloWorld
  {
    static void Main(string[] args)
    {
      /* my first program in C# */
      Console.WriteLine("Hello World");
      Console.ReadKey();
    }
  }
}
```

C# - Program Structure

- The first line of the program using System; - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.

- The next line has the namespace declaration. A namespace is a collection of classes. The HelloWorldApplication namespace contains the class HelloWorld.

- WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

- The last line Console.ReadKey(); is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

C# - Program Structure

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, file name could be different from the class name.

C# - Program Structure

- You can compile a C# program by using the command line instead of the Visual Studio IDE:
- Open a text editor and add the above mentioned code.
- Save the file as helloworld.cs
- Open the command prompt tool and go to the directory where you saved the file.
- Type csc helloworld.cs and press enter to compile your code.
- If there are no errors in your code the command prompt will take you to the next line and would generate helloworld.exe executable file.
- Next, type helloworld to execute your program.
- You will be able to see "Hello World" printed on the screen.

C# - Basic Syntax

- **The using Keyword**
- using System;
- The using keyword is used for including the namespaces in the program. A program can include multiple using statements.
- **The class Keyword**
- The class keyword is used for declaring a class.
- **Comments in C#**
- The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:
- /* This program demonstrates The basic syntax of C# programming Language */
- Single line comments are indicated by the '//' symbol. For example,
- }//end class Rectangle

C# - Basic Syntax

- **Member Variables**
- Variables are attributes or data members of a class, used for storing data.
- **Member Functions**
- Functions are set of statements that perform a specific task. The member functions of a class are declared within the class
- **Instantiating a Class**
- In the program, the class XYZ is used as a class which contains the Main() method and instantiates the other class.
- **Identifiers**
- An identifier is a name used to identify a class, variable, function, or any other user-defined item.

C# - Basic Syntax

- **The basic rules for naming classes in C# are as follows:**
- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore.
- The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However an underscore (_) can be used.
- It should not be a C# keyword.

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get

C# - Data Types

- **In C#, variables are categorized into the following types:**
- Value types
- Reference types
- Pointer types

- **Value Types**
- Value type variables can be assigned a value directly. They are derived from the class System.ValueType.
- The value types directly contain data. Some examples are int, char, float, which stores numbers, alphabets and floating point numbers respectively. When you declare an int type, the system allocates memory to store the value.

C# - Value Types

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0

C# - Value Types

Type	Represents	Range	Default Value
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit signed integer type	0 to 4,294,967,295	0
ulong	64-bit signed integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit signed integer type	0 to 65,535	0

C# - Value Types

- To get the exact size of a type or a variable on a particular platform, you can use the sizeof method.
- The expression sizeof(type) yields the storage size of the object or type in bytes.
- The table lists data types are the available value types in C# 2010.

C# - Reference Types

- The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.
- Using more than one variable, the reference types can refer to a memory location.
- If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value.
- The user defined reference types are:

  class, interface, or delegate (Discuss in later).
- Example of built in reference types are: object, dynamic and string.

C# - Reference Types

- **Object Type**
- The Object Type is the ultimate base class for all data types in C# Common Type System(CTS).
- Object is an alias for System.Object class. So object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.
- However, before assigning values, it needs type conversion.
- When a value type is converted to object type, it is called boxing and on the other hand, when an object type is converted to a value type it is called unboxing.

```
Object obj;
obj = 100; // this is boxing
```

C# - Reference Types

- **Dynamic Type**
- You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at runtime.
- Syntax for declaring a dynamic type is:

  dynamic <variable_name> = value;
- For example,

  dynamic d = 20;
- Dynamic types are similar to object types except that, type checking for object type variables takes place at compile time, whereas that for the dynamic type variables take place at run time.

C# - Reference Types

- **String Type**
- The String Type allows you to assign any string values to a variable.
- The string type is an alias for the System.String class.
- It is derived from object type.
- The value for a string type can be assigned using string literals in two forms: quoted and @quoted.
- For example,

  String str = “Hello World";
- A @quoted string literal looks like:

  @“Hello World";

C# - Pointer Types

- Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as in C or C++.

- Syntax for declaring a pointer type is:

  type* identifier;

- For example,

  char* cptr;

  int* iptr;

C# - Type Conversion

- Type conversion is basically type casting, or converting one type of data to another type. In C#, type casting has two forms:

- Implicit type conversion - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- Explicit type conversion - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

C# - Type Conversion

```
namespace TypeConversionApplication
{
        class ExplicitConversion
      {
            static void Main(string[] args)
             {
                  double d = 5673.74;
                 int i;            // cast double to int.
                 i = (int)d;
                   Console.WriteLine(i);
  Console.ReadKey();
              }
        }
}
```

O/p: 5673

C# - Type Conversion

- **C# provides the following built-in type conversion methods:**
- ToBoolean
- ToByte
- ToChar
- ToDateTime
- ToDecimal
- ToDouble
- ToInt16
- ToInt32
- ToInt64
- ToSbyte
- ToSingle
- ToType
- ToUInt16
- ToUInt32
- ToUInt64

C# - Variables

- The basic value types provided in C# can be categorized as:

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

Nullable Types

- In C#, the compiler does not allow you to assign a null value to a variable. So, C# 2.0 introduced the Nullable type.
- The Nullable type allows you to assign a null value to a variable. IT can only work with Value Type, not with Reference Type.
- The Nullable type is an instance of System.Nullable<T> struct. Here T is a type which contains non-nullable value types like integer type, floating-point type, a boolean type, etc.
- For example, in nullable of integer type you can store values from -2147483648 to 2147483647, or null value.
- **Syntax**

  Nullable<data_type> variable_name = null;

  Or you can also use a shortcut which includes ? Operator with the data type. E.g.

  datatype? variable_name = null;

Nullable Types

Example:

```
// this will give compile time error
 int i = null;

// Valid declaration
Nullable<int> j = null;

// Valid declaration
 int? k = null;
```

- You cannot directly access the value of the Nullable type. You have to use GetValueOrDefault() method to get an original assigned value if it is not null. E.g.

```
Console.WriteLine(k.GetValueOrDefault());
```

Nullable Types

- **Advantage of Nullable Types:**
- The main use of nullable type is in database applications. Suppose, in a table a column required null values, then you can use nullable type to enter null values.
- Nullable type is also useful to represent undefined value.
- You can also use Nullable type instead of a reference type to store a null value.

C# - Variables

- Variable Declaration in C#
- Syntax for variable declaration in C# is:
  <data_type> <variable_list>;

- Examples:

```
    int i, j, k;
char c, ch;
float f, salary;
double d;
```

C# - Variables

- Variable Initialization in C#
- Syntax of variable initialization
  <data_type> <variable_name> = value;

- Some examples are:
  int d = 3, f = 5; /* initializing d and f. */
  byte z = 22; /* initializes z. */
  double pi = 3.14159; /* declares an approximation of pi. */
  char x = 'x'; /* the variable x has the value 'x'*/

C# - Variables

- Accepting Values from User

- The Console class in the System namespace provides a function ReadLine() for accepting input from the user and store it into a variable.

- For example,

  int num;

  num = Convert.ToInt32(Console.ReadLIne());

C# - Variables

- **Lvalues and Rvalues in C#:**
- There are two kinds of expressions in C#:
- **lvalue** : An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** : An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

C# - Constants and Literals

- The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.
- The constants are treated just like regular variables except that their values cannot be modified after their definition.

• Types of Constants in C#

- Integer Literals
- Floating-point Literals
- Character Constants
- String Literals

C# - Constants and Literals

- Defining Constants
- Constants are defined using the const keyword.

- Syntax for defining a constant is:
- const <data_type> <constant_name> = value;

- Example:
- const double pi = 3.14159; // constant declaration

C# - Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

- C# provides the following type of operators:
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

C# - Operators

- Arithmetic Operators

  ++ and –- also can be consider as Arithmatic operator

- Bitwise Operators

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

C# - Operators

- Assume if A = 60; and B = 13; Now in binary format they will be as follows:

  A = 0011 1100

  B = 0000 1101

  A&B = 0000 1100 (Bitwise AND)

  A|B = 0011 1101 (Bitwise OR)

  A^B = 0011 0001 (Bitwise XOR)

  ~A = 1100 0011 (Binary 1's Compl.)

C# - Operators

- **Misc Operators**

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of a variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Operators Precedence in C#

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

C# - Loops

Loop Type	Description
while loop	Repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

C# - Encapsulation

- Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'.
- Encapsulation, in object oriented programming methodology, prevents access to implementation details.
- Abstraction and encapsulation are related features in object oriented programming.
- Abstraction allows making relevant information visible and encapsulation enables a programmer to implement the desired level of abstraction.
- Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member.

C# : access specifiers

Public	Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.
Private	Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.
Protected	Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance.

C# : access specifiers

Internal	Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.
Protected internal	The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

C# - Class Methods

- A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.
- To use a method, you need to:
- Define the method
- Call the method

- Defining Methods in C#

<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
Method Body
}

Passing Parameters to a Method

Mechanism	Description
Value parameters	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Reference parameters	This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
Output parameters	This method helps in returning more than one value.

C# - Passing Parameters by Reference

```
class NumberManipulator {
 public void swap(ref int x, ref int y)      {
          //Body
       }
      static void Main(string[] args)      {
         NumberManipulator n = new NumberManipulator();
         /* local variable definition */
         int a = 100;
         int b = 200;
          /* calling a function to swap the values */
         n.swap(ref a, ref b);
     }
}
```

C# - Passing Parameters by Output

```
public void getValues(out int x, out int y )     {

 Console.WriteLine("Enter the first value: ");
 x = Convert.ToInt32(Console.ReadLine());
 Console.WriteLine("Enter the second value: ");
 y = Convert.ToInt32(Console.ReadLine());
}

 int a , b;
/* calling a function to get the values */ n.getValues(out a,
 out b);
```

C# - Arrays

- **Declaring Arrays**

  datatype[] arrayName;
- **Initializing an Array**

  double[] balance = new double[10];

  int [] marks = new int[5] { 99, 98, 92, 97, 95};

C# - Param Arrays

Concept	Description
Multi-dimensional arrays	C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Jagged arrays	C# supports multidimensional arrays, which are arrays of arrays.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Param arrays	This is used for passing unknown number of parameters to a function.
The Array Class	Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

C# - Multidimensional Arrays

- You can declare a **2 dimensional** array of strings as:

```
string [,] names;
```

- or, a **three dimensional** array of int variables:

```
int [ , , ] m;
```

- **Initializing Two-Dimensional Arrays**

```
int [,] a = int [3,4] = { {0, 1, 2, 3} , /* row 0 */
                          {4, 5, 6, 7} , /* row 1 */
                          {8, 9, 10, 11} /*row 2 */ };
```

- **Accessing Two-Dimensional Array Elements**

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
  Console.WriteLine(numbers[0, 2]); //output: 2
```

C# - Jagged Arrays

- **Declare a jagged array**

  int [][] scores; OR

  data_type[][] name = new data_type[row][];

  Int[][] jagged_arr = new int[4][];

- **Initialize a jagged array**

  int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};

- Where, scores is an array of two arrays of integers -- scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

C# - Jagged Arrays

- **Initialize a jagged array**

```
int[][] scores = new int[2][]{new int[]{92,93,94},new
                 int[]{85,66,87,88}};
```

- Where, scores is an array of two arrays of integers -- scores[0] is an array of 3 integers and scores[1] is an array of 4 integers. OR

- int [][] jagged_arr = **new int[][]**

```
    {
    new int[] {9, 2, 3, 3},
    new int[] {1, 2, 4, 3},
    new int[] {6, 5, 3, 7},
    new int[] {1, 2, 3, 4}
    }
```

C# - Passing Arrays as Function Arguments

```
class MyArray
  {
     double getAverage(int[] arr, int size)
     {

     }
     static void Main(string[] args)
     {
        MyArray app = new MyArray();
        /* an int array with 5 elements */
        int [] balance = new int[]{1000, 2, 3, 17, 50};
        /* pass pointer to the array as an argument */
        avg = app.getAverage(balance, 5 ) ;
     }
}
```

C# - Params Arrays

- The params keyword creates an array at runtime that receives and holds n number of parameters.
- The "params" keyword in C# allows a method to accept a variable number of arguments. C# params works as an array of objects.
- By using params keyword in a method argument definition, we can pass a number of arguments.
- static int add(params int[] allnumber)

C# - Params Arrays

```
using System;namespace ArrayApplication {
   class ParamArray {
         public int AddElements(params int[] arr) {
               int sum = 0;
         foreach (int i in arr) {
               sum += i;
         }
         return sum;
         }
   }
   class TestClass {
   static void Main(string[] args) {
         ParamArray app = new ParamArray();
         int sum = app.AddElements(512, 720, 250, 567, 889);
         Console.WriteLine("The sum is: {0}", sum);
         Console.ReadKey();
   }
   }
}
```

C# - Array Class

- It is defined in the System namespace.

Property Name & Description
IsFixedSize Gets a value indicating whether the Array has a fixed size.
IsReadOnly Gets a value indicating whether the Array is read-only.
Length Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
LongLength Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
Rank Gets the rank (number of dimensions) of the Array.

C# - Param Arrays

Method Name & Description
Clear Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
Copy(Array, Array, Int32) Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
CopyTo(Array, Int32) Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
GetLength Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
GetLongLength

C# - Param Arrays

Method Name & Description
GetLowerBound Gets the lower bound of the specified dimension in the Array.
GetType Gets the Type of the current instance. (Inherited from Object.)
GetUpperBound Gets the upper bound of the specified dimension in the Array.
GetValue(Int32) Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
IndexOf(Array, Object) Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.
Reverse(Array) Reverses the sequence of the elements in the entire one-dimensional Array.

C# - Param Arrays

Method Name & Description
SetValue(Object, Int32) Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
Sort(Array) Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.
ToStringk Returns a string that represents the current object. (Inherited from Object.)

C# – Namespaces

- A namespace is designed for providing a way to keep one set of names separate from another.
- **Defining a Namespace**

```
namespace namespace_name
{
 // code declarations
}
```

- **Nested Namespaces**

```
namespace namespace_name1
{   // code declarations
    namespace namespace_name2
                { // code declarations }
}
```

C# – Namespaces

```
using System;
namespace first_space {
  class namespace_cl
  {
    public void func()
    {    }
  }
}
namespace second_space {
  class namespace_cl
  {
    public void func()
    {    }
```

C# – Namespaces

```
class TestClass
{
  static void Main(string[] args)
  {
    first_space.namespace_cl fc = new
        first_space.namespace_cl();
    second_space.namespace_cl sc = new
        second_space.namespace_cl();
    fc.func();     sc.func();
  }
}
```

C# - Strings

- **string** keyword to declare a string variable. The string keyword is an alias for the **System.String** class.
- **Creating a String Object**
- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or object to its string representation

C# - Strings

Property Name & Description
Chars Gets the *Char* object at a specified position in the current *String* object. **Chars property is an indexer and is implicit in C#.**
Length Gets the number of characters in the current String object.

- There is no technical difference between string and String. In C# string is an alias for System.String.
- It is recommended to use string as it works even without "using System;".

C# - Strings

Method Name & Description
public bool Contains(string value) Returns a value indicating whether the specified string object occurs within this string.
public static bool IsNullOrEmpty(string value) Indicates whether the specified string is null or an Empty string.
public static string Join(string separator, params string[] value) Concatenates all the elements of a string array, using the specified separator between each element.
public static string Join(string separator, string[] value, int startIndex, int count) Concatenates the specified elements of a string array, using the specified separator between each element.
public string[] Split(params char[] separator) Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array.

C# - Strings

- **Length of the string**

```
string str = "C# Programming";
int length = str.Length;
Console.WriteLine("Length: "+ length);
```

Output: 14

- **Compare two strings**

```
string str1 = "C# Programming";
string str2 = "C# Programming";
string str3 = “Visual Studio";
Boolean result1 = str1.Equals(str2);
Console.WriteLine("string str1 and str2 are equal: " + result1);
Boolean result2 = str1.Equals(str3);
Console.WriteLine("string str1 and str3 are equal: " + result2);
```

Output:

True

False

C# - Strings

- **Join two strings**

```
string str1 = "C# ";
String str2 = “Programming”;
string joinedString = string.Concat(str1, str2);
Console.WriteLine("Joined string: " + joinedString);
```

Output: Joined string: C# Programming

- **String interpolation**
- In C#, we can use string interpolation to insert variables inside a string. For string interpolation, the string literal must begin with the $ character. For example,

```
string name = “DDU"; // string interpolation
string message = $"Welcome to {name}";
Console.WriteLine(message);
```

Output: Welcome to DDU

C# - Strings

- **Immutability of String Objects**
- In C#, strings are immutable. This means, once we create a string, we cannot change that string. E.g.,

```
string str = "Hello ";
// add another string "World"
// to the previous string example
str = string.Concat(str, "World");
```

- How are we able to modify the string when they are immutable?
  - C# takes the value of the string "Hello “.
  - Creates a new string by adding "World" to the string "Hello “.
  - Creates a new string object, gives it a value "Hello World", and stores it in str.
  - The original string, "Hello ", that was assigned to str is released for garbage collection because no other variable holds a reference to it.

C# - Enums

- An enumeration is a set of named integer constants.
- C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.
- **Declaring enum Variable**

```
enum <enum_name>
{
        enumeration list
};
```

Where,

- The enum_name specifies the enumeration type name.
- The enumeration list is a comma-separated list of identifiers.

C# - Enums

- **Example**

  enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

  where,

  int Monday = (int)Days.Mon;

  int Friday = (int)Days.Fri;

- **values**

  Monday: 1

  Friday: 5

C# - Classes

```
<access specifier> class  class_name
{   // member variables
 <access specifier> <data type> variable1;     <access specifier>
   <data type> variable2;     ...     <access specifier> <data type>
   variableN;

// member methods
<access specifier> <return type> method1(parameter_list)
   {       // method body
    }

}
```

C# - Classes

- Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is internal. Default access for the members is private.
- Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.
- To access the class members, you will use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

C# - Classes

- **Constructors in C#**
- A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
- A constructor will have exact same name as the class and it does not have any return type
- A default constructor does not have any parameter but if you need a constructor can have parameters. Such constructors are called parameterized constructors.

C# - Classes

- **Destructors in C#**
- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope.
- A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

```
class Line   {
  public Line()  // constructor
  {
     Console.WriteLine("Object is being created");
  }
  ~Line() //destructor
  {
```

C# - Classes

- **Static Members of a C# Class**
- We can define class members as static using the static keyword.
- When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- The keyword static implies that only one instance of the member exists for a class.
- Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.

C# - Classes

- Static variables can be initialized outside the member function or class definition.
- You can also initialize static variables inside the class definition.

- You can also declare a member function as static. Such functions can access only static variables.
- The static functions exist even before the object is created.

C# - Inheritance

- The idea of inheritance implements the IS-A relationship.
- For example, car IS A Vehicle,
- Swift IS-A car
- hence swift IS-A Vehicle as well and so on.
- Syntax

```
<acess-specifier> class <base_class>
{ ...
}
class <derived_class> : <base_class>
{ ...
}
```

C# - Inheritance

- **Base Class Initialization**

```
class Rectangle   {
     protected double length;
     protected double width;
     public Rectangle(double l, double w)
     {
        length = l;  width = w;
     }
  }//end class Rectangle
class Tabletop : Rectangle
 {     private double cost;
          public Tabletop(double l, double w) : base(l, w)
          { }
  }
```

Abstract Class

- In C#, an abstract class is a class that cannot be instantiated. Instead, it serves as a base class for other classes to inherit from.
- Abstract classes are used to define a common set of behaviors or properties that derived classes should have.
- To create an abstract class in C#, you use the “abstract” keyword before the class definition, which indicates incomplete implementation.
- Abstraction hides the internal details and show only the functionality.
- The keyword **abstract** is used to declare the class or method as abstract. Also, the **abstract** modifier can be used with **indexers**, **events**, and **properties**.

Abstract Class

- Abstract class can also contain non-abstract methods also.
- **Abstract methods**
- A method that does not have a body is known as an abstract method. We use the abstract keyword to create abstract method. E.g.

  public abstract void display();
- Here, display() is an abstract method. An abstract method can only be present inside an abstract class.
- When a non-abstract class inherits an abstract class, it should provide an implementation of the abstract methods.

Sealed Class

- Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the ***sealed*** keyword.
- The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class. E.g.

```
sealed class class-name{
.....
}
```

- *A method can also be sealed*, and in that case, the method cannot be overridden. However, a method can be sealed in the classes in which they have been inherited.
- If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.

Sealed Class

- Local variables can't be sealed.
- The main purpose of the sealed class is to withdraw the inheritance attribute from the user so that they can’t attain a class from a sealed class.
- Sealed classes are used best when you have a class with static members.

C# - Interface

- **C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance.
- **Declaring Interfaces**

```
public interface ITransactions
{
   // interface members
   void showTransaction();
   double getAmount();
}
public class Transaction : ITransactions
 { }
```

C# - Interface

- In Multiple inheritance we can use interface like

```
class Rectangle : Shape, PaintCost
{     }
```

- Where Shape is Base Class and PaintCost is an interface

C# - Polymorphism

- In static polymorphism the response to a function is determined at the compile time.
- In dynamic polymorphism it is decided at run time.
- **Static Polymorphism**
- **Function overloading**
- **Operator overloading**
- **Dynamic Polymorphism**
- Dynamic polymorphism is implemented by abstract classes and virtual functions.
- When you have a function defined in a class that you want to be implemented in an inherited class(es), you use virtual functions.
- The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

C# Dynamic Polymorphism

- **abstract classes:**
  - You cannot create an instance of an abstract class
  - You cannot declare an abstract method outside an abstract class
  - When a class is declared sealed, it cannot be inherited, abstract classes cannot be declared sealed.

C# Dynamic Polymorphism

```
abstract class Shape
 {
   public abstract int area();
 }
 class Rectangle:  Shape
 {
   private int length;
   private int width;
   public override int area ()
   {
     Console.WriteLine("Rectangle class area :");
     return (width * length);
   }
 }
```

C# Dynamic Polymorphism

```
class Shape   {
    protected int width, height;
    public virtual int area()     {
      Console.WriteLine("Parent class area :");
      return 0;
    }
  }
  class Rectangle: Shape
  {
    public override int area ()
    {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
```

C# - Operator Overloading

- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined.

```
public static Box operator+ (Box b, Box c){
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
static void Main(string[] args) {
    Box Box1 = new Box();                  // Declare Box1 of type Box
    Box Box2 = new Box();                  // Declare Box2 of type Box
    Box Box3 = new Box();                  // Declare Box3 of type Box
    Box3 = Box1 + Box2;                    // volume of box 3
 }
```

C# - Operator Overloading

- **Overloadable and Non-Overloadable Operators**

Operators	Description
+, -, !, ~, ++, --	These unary operators take one operand and can be overloaded.
+, -, *, /, %	These binary operators take one operand and can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded directly.
+=, -=, *=, /=, %=	The assignment operators cannot be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded.

Type Safety

- Type safety in .NET has been introduced to prevent the objects of one type from peeking into the memory assigned for the other object.
- Example

```
class TSafe    {
            public int tsafe { get; set; }
}
class TSafe1  {
      public int tsafe { get; set; }
      public int tsafe2 { get; set; }
}
class test {
      static void Main(string[] args)       {
         TSafe ts = new TSafe();
            TSafe1 ts1 = (TSafe1) ts;                  //Error
            }
}
```

Type Safety

- In memory tsafe would be referencing the 4 bytes of space and suppose next to that part of memory is another string.
- Now suppose we cast object ts to TSafe1 that thankfully is not at all possible in .NET, but for a moment imagine that it would have been possible.
- So the last line will throw a compile time error:

  Error: Cannot convert type 'TypeSafety.TSafe' to 'TypeSafety.TSafe1'

Comparing for Equality

- Both the == Operator and the Equals() method are used to compare two value type data items or reference type data items.
- The Equality Operator (==) is the comparison operator and the Equals() method compares the contents of a string.
- The == Operator compares the reference identity while the Equals() method compares only contents.

Comparing for Equality

- Example

```
string str1 = "Hello“;
char[] str2 = { 'H', 'e', 'l', 'l', 'o' };
object str3 = new string(str2);
Console.WriteLine("With == comparison:" + (str1==str3));
Console.WriteLine("Comparison with equals()" + (str1.Equals(str3)));
```

- **Output:**

```
Comparison With == operator : False
Comparison with equals() :True
```

- If we change (str1==str3) to (str1== (string) str3) the result will be true as it will do value comparison rather than reference comparison as in case of the above code.

User-Define Casts

- When writing custom, Implicit conversion operators are those that don't require an explicit cast. Explicit conversion operators are those that do require an explicit cast.
- In C#, implicit and explicit operators are used to convert one data type to another.
- **Implicit operators** are used when the conversion is guaranteed to succeed without data loss.
- Implicit operators are used to make the code more concise and readable and when the conversion is frequently performed.
- **Explicit operators** are used when the conversion might result in data loss or when the conversion is not guaranteed to succeed.
- Explicit operators are used to avoid unexpected behaviour or to handle exceptional cases.

User-Defined Casts

- **Overloading Implicit and Explicit Operators in C#**
- In C#, we can overload the implicit and explicit operators for our custom types.
- To overload the implicit and explicit operators, we need to define them as static methods in our custom type. Here's the syntax.

```
public static implicit/explicit operator TargetType (SourceType source)
{
        // conversion logic here
 }
```

- The implicit and explicit keywords indicate the type of operator we're overloading. The SourceType is the type we're converting from, and the TargetType is the type we're converting to.

User-Defined Casts: Example

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

namespace ImplicitExplicit {
   class Temperature    {
      public float Degrees { get; set; }
      public static void Main()       {
         Celsius cel = new Celsius(10);
         Fahrenheit far = cel;
         Celsius cel2 = far;
         Console.WriteLine(cel2.Degrees);
         Console.WriteLine(far.Degrees);
         Console.ReadKey();
      }
   }
```

User-Defined Casts

```
class Celsius : Temperature    {
    public Celsius(float temp)       {
        Degrees = temp;
    }
    public static implicit operator Fahrenheit(Celsius c)       {
        return new Fahrenheit((9.0f / 5.0f) * c.Degrees + 32);
    }
  }
  class Fahrenheit : Temperature    {
    public Fahrenheit(float temp)       {
        Degrees = temp;
    }
    public static implicit operator Celsius(Fahrenheit fahr)       {
        return new Celsius((5.0f / 9.0f) * (fahr.Degrees - 32));
    }
  }
}
```

Memory Management

- C# manages memory in two distinct regions: the stack and the heap.
- The stack is used for storing value types, method parameters, and local variables, while the heap is used for storing reference types (objects).
- The garbage collector is responsible for managing memory in the heap.
- Value types (e.g., int, float, structs) are stored directly on the stack, while reference types (e.g., classes, arrays) are stored on the heap.
- When you assign a value type, a copy of the value is created, whereas assigning a reference type creates a new reference to the same object.

Memory Management

- A **value type** holds the data within its own memory location.
- Value types => *bool, byte, char, decimal, double, float, int, long, uint, ulong, ushort, enum, struct*
- A **reference type** contains a pointer to another memory location that holds the real data.
- Reference types => *class, interface, delegate, string, object, dynamic, arrays*
- **Automatic Memory Management**
- C# automatically manages memory allocation and deallocation, leveraging the garbage collector to handle memory used by reference types.
- As a developer, you don't need to worry about explicitly releasing memory, as the GC will handle it for you.

Freeing Unmanaged Resources

- **Managed resources** basically means "managed memory" that is managed by the garbage collector. When you no longer have any references to a managed object (which uses managed memory), the garbage collector will (eventually) release that memory for you.
- **Unmanaged resources** are then everything that the garbage collector does not know about. For example:
  - Open files
  - Open network connections
  - Unmanaged memory
  - In XNA: vertex buffers, index buffers, textures, etc.
  - Normally you want to release those unmanaged resources before you lose all the references you have to the object managing them.

Freeing Unmanaged Resources

- The following are two mechanisms to automate the freeing of unmanaged resources:

1. Declaring a destructor (or Finalizer) as a member of your class.
2. Implementing the System.IDisposable interface in your class. (The IDisposable interface consists of only one Dispose method with no arguments)

- **Finalization** is the process by which the GC allows objects to clean up any unmanaged resources that they're holding, before actually destroying the instance.
- An implementation of the Finalize method is called a "finalizer." Finalizers should free only external resources held directly by the object itself.
- The GC attempts to call finalizers on objects when it finds that the object is no longer in use—when no other object is holding a valid reference to it.

Freeing Unmanaged Resources

- Unlike Finalize, developers should call **Dispose** explicitly to free unmanaged resources.
- You should call the Dispose method explicitly on any object that implements it to free any unmanaged resources for which the object may be holding references.
- Dispose doesn't remove the object itself from memory. The object will be removed when the garbage collector finds it convenient.
- It should be noted that the developer implementing the Dispose method must call GC.SuppressFinalize(this) to prevent the finalizer from running.

C# - Unsafe Codes

- C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier.
- The **unsafe code** or the unmanaged code is a code block that uses a pointer variable.
- Instead of declaring an entire method as **unsafe**, you can also declare a part of the code as unsafe. Ex.

```
static unsafe void Main(string[] args)
{
        int var = 20;
        int* p = &var;
        Console.WriteLine("Data is: {0} ",  var);
        Console.WriteLine("Address is: {0}",  (int)p);
        Console.ReadKey();
}
```

C# - Unsafe Codes

- You can retrieve the data stored at the located referenced by the pointer variable, using the **ToString()** method.

```
public static void Main()
{
   unsafe
   {
      int var = 20;
      int* p = &var;
      Console.WriteLine("Data is: {0} " , var);
      Console.WriteLine("Data is: {0} " , p->ToString());
      Console.WriteLine("Address is: {0} " , (int)p);
   }

}
```

C# - Unsafe Codes

Compiling Unsafe Code

- For compiling unsafe code, you have to specify the /unsafe command line switch with command line compiler.
- For example, to compile a program named prog1.cs containing unsafe code, from command line, give the command: csc /unsafe prog1.cs
- If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.
    1. Open **project properties** by double clicking the properties node in the **Solution Explorer**.
    2. Click on the **Build tab**.
    3. Select the **option** "Allow unsafe code".

C# - Unsafe Codes

```
int[] list = {10, 100, 200};
fixed(int *ptr = list)

/* let us have array address in pointer */
for ( int i = 0; i < 3; i++)        {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
        }
```

Advantages and Disadvantages

- **Advantages of Unsafe Mode**
- It increases the performance of the Program.
- We use fixed buffers inside an unsafe context.
- With a fixed buffer, you can write and read raw memory without any of the managed overhead.
- It provides a way to interface with memory.
- **Disadvantages of Unsafe Mode**
- It increases the responsibility of the programmer to check for security issues and extra developer care is paramount to averting potential errors or security risks.
- It bypasses security. Because the CLR maintains type safety and security, C# does not support pointer arithmetic in managed code, unlike C/C++.
- The unsafe keyword allows pointer usage in unmanaged code but safety is not guaranteed because strict object access rules are not followed.
- It also avoids type checking, that can generate errors sometimes.

When to use Unsafe Mode

- If we are using pointers.
- When code interfaces with the operating system or other unmanaged code.
- If we want to implement a time-critical algorithm or want to access a memory-mapped device.

C# – Exception Handling

- The exception classes in C# are mainly directly or indirectly derived from **the System.Exception** class.
- Some of the exception classes derived from the **System.Exception** class are the **System.ApplicationException** and **System.SystemException** classes.
- The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should be derived from this class.
- The **System.SystemException** class is the base class for all predefined system exception.

C# - Exception Handling

- An exception is a problem that arises during the execution of a program.
- Syntax

```
fry {
   // statements causing exception
}
catch( ExceptionName e1 ) {
   // error handling code
}
catch( ExceptionName e2 ) {
   // error handling code
}
Finally {
   // statements to be executed
}
```

C# – Exception Handling

- Some of the predefined exception classes derived from the Sytem.SystemException class

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchExcepti on	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.

C# – Exception Handling

- User defined exception classes are derived from the **ApplicationException** class.

```
public class TempIsZeroException: ApplicationException
{
   public TempIsZeroException(string message):
    base(message)
   {
   }
}
public class Temperature {
   int temperature = 0;
   public void showTemp()    {
      if(temperature == 0)       {
         throw (new TempIsZeroException("Zero Temperature
found"));
      }
   }
}
```

C# – Exception Handling

- You can throw an object if it is either directly or indirectly derived from the System.Exception class.
- You can use a throw statement in the catch block to throw the present object

```
catch(Exception e)
{
    ...
   Throw e
}
```

C# – Delegates

- A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter?
- How does C# handle the callback functions or event handler? The answer is – delegate.
- C# delegates are similar to pointers to functions, in C or C++.
- A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods.
- All delegates are implicitly derived from the System.Delegate class.

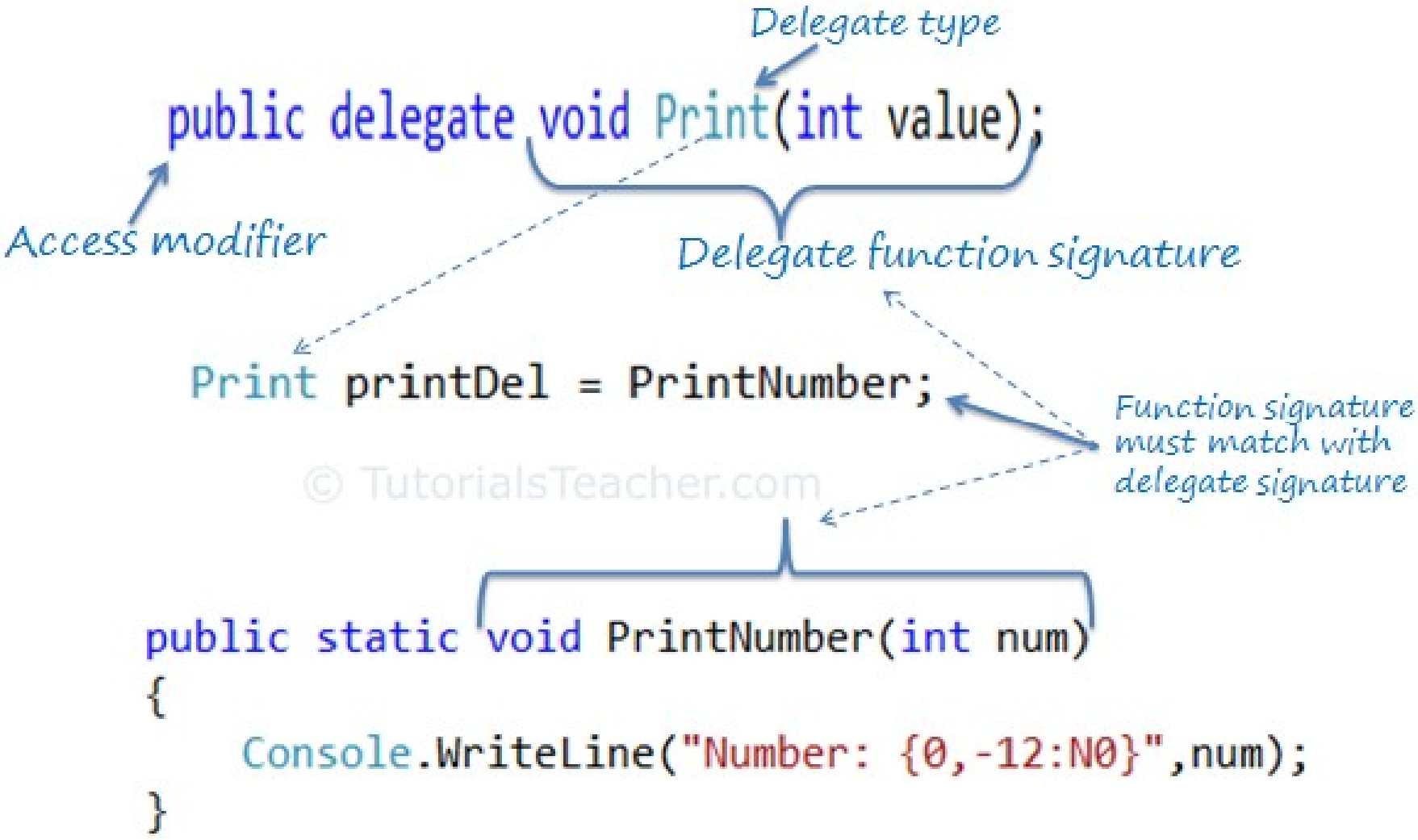
- Syntax

delegate <return type> <delegate-name><parameter list>

C# – Delegates

- **Declaring Delegates**
- Delegate declaration determines the methods that can be referenced by the delegate.
- A delegate can refer to a method, which have the same signature as that of the delegate.

- **Points to Remember :**
- Delegate is a function pointer. It is reference type data type.
- Syntax: public delegate void <function name>(<parameters>)
- A method that is going to be assigned to delegate must have same signature as delegate.
- Delegates can be invoke like a normal function or Invoke() method.
- Multiple methods can be assigned to the delegate using "+" operator. It is called multicast delegate.


Delegate type
public delegate void Print(int value);
Access modifier
Delegate function signature
Print printDel = PrintNumber;
Function signature must match with delegate signature
© TutorialsTeacher.com
public static void PrintNumber(int num)
{
Console.WriteLine("Number: {0,-12:N0}",num);
}

C# – Delegates

- **Instantiating Delegates**

```
public delegate void printString(string s);
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
      public static int AddNum(int p)
      {
         num += p;
         return num;
      }
```

C# - Delegates

```
    public static int MultNum(int q)        {
                num *= q;
               return num;
    }
    public static int getNum()              {
               return num;
    }
static void Main(string[] args)
{
    //create delegate instances
    NumberChanger nc1 = new NumberChanger(AddNum);
    NumberChanger nc2 = new NumberChanger(MultNum);
    //calling the methods using the delegate objects
    nc1(25);   // gives 35
    Console.WriteLine("Value of Num: {0}", getNum());
    nc2(5);    //gives 175
 }
```

C# – Delegates

- **Multicasting of a Delegate**
- Delegate objects can be composed using the "+" operator.
- A composed delegate calls the two delegates it was composed from.
- Only delegates of the same type can be composed. The "–" operator can be used to remove a component delegate from a composed delegate.
- Using this useful property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called multicasting of a delegate.

C# – Delegates

- **Multicasting of a Delegate**
- It is possible for certain delegates to hold and invoke multiple methods. Such delegates are called multicast delegates. Multicast delgates also known as combinable delegates, must satisfy the following conditions:
- The return type of delegate must be void.
- None of the parameters of the delegate type can be declared as output parameter.

C# - Multicasting of a Delegate

```
static void Main(string[] args)
{
        //create delegate instances
        NumberChanger nc;
        NumberChanger nc1 = new NumberChanger(AddNum);
        NumberChanger nc2 = new NumberChanger(MultNum);
        nc = nc1;
        nc += nc2;
        //calling multicast
        nc(5); //1st call nc (nc1) and then nc2
        Console.WriteLine("Value of Num: {0}", getNum());
}
 //output: Value of Num: 75
```

C# – Delegates

- **Use of Delegate**

```
namespace DelegatesDemo{
    class PrintString    {
        static FileStream fs;
        static StreamWriter sw;
        // delegate declaration
        public delegate void printString(string s);

        // this method prints to the console
        public static void WriteToScreen(string str)
        {
            Console.WriteLine("The String is: {0}", str);
        }
```

C# - Delegates

```
 //this method prints to a file
public static void WriteToFile(string s)       {
          fs = new FileStream("H:\\message.txt",
          FileMode.Append, FileAccess.Write);
          sw = new StreamWriter(fs);
          sw.WriteLine(s);
          sw.Flush();
          sw.Close();
          fs.Close();
       }
// this method takes the delegate as parameter and uses it to
       public static void sendString(printString ps) {
          ps("Hello World");
       }
```

C# - Delegates

```
 static void Main(string[] args)       {
          printString ps1 = new printString(WriteToScreen);
          printString ps2 = new printString(WriteToFile);
          sendString(ps1);
          sendString(ps2);
          Console.ReadKey();
       }
   }
}
```

Output: The String is: Hello World

And it also create a message.txt file and print “Hello World” in it

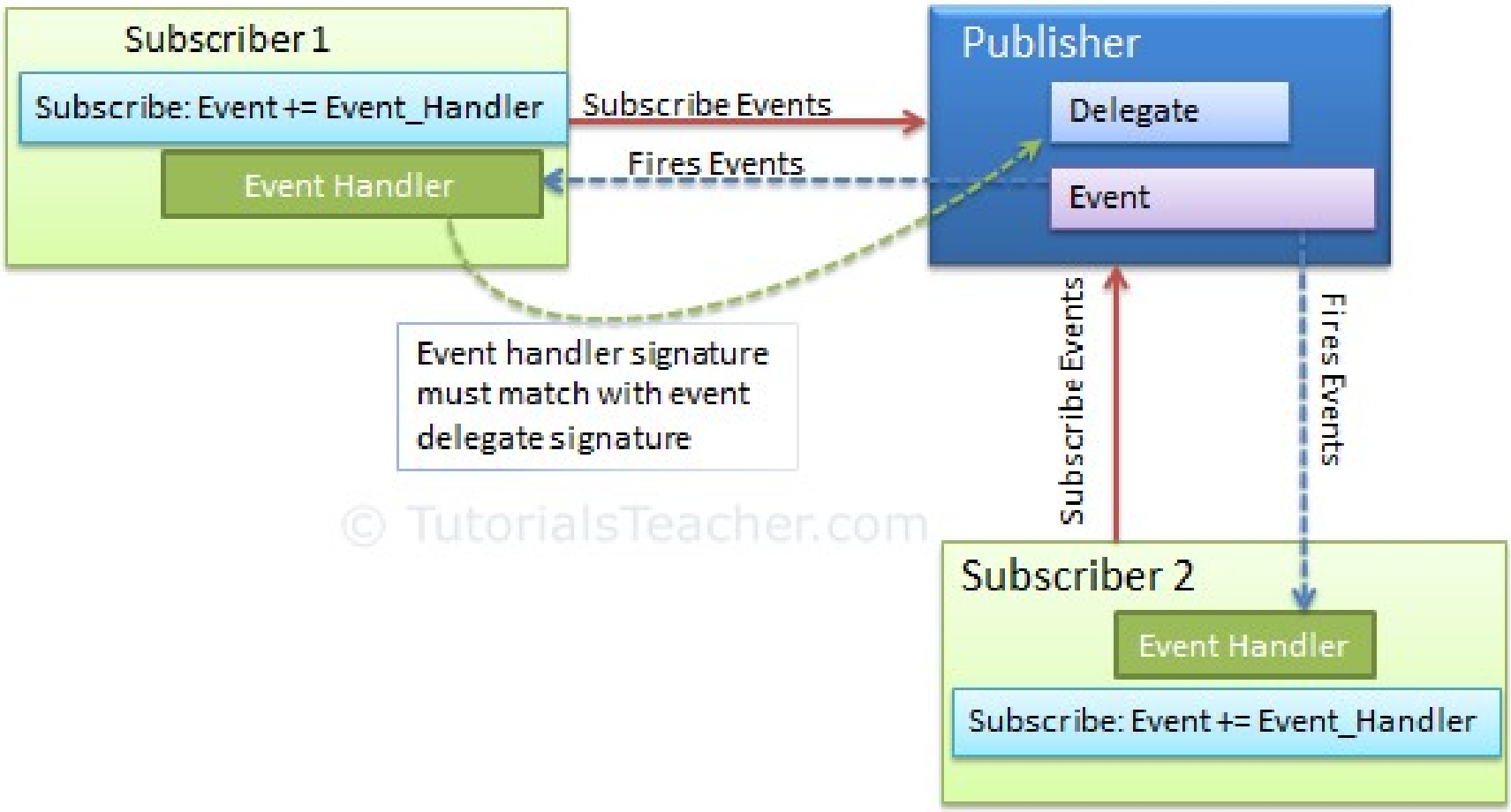
C# – Events

- **Publisher-Subscriber-Notification-Handler**
- **Events** are basically a user action like key press, clicks, mouse movements etc., or some occurrence like system generated notifications.
- UI controls use events extensively. (For example, the button control in a Windows form has multiple events such as click, mouseover, etc.)
- Applications need to respond to events when they occur. For example, interrupts.
- Events are used for inter-process communication.

C# – Events

- **Using Delegates with Events**
- An event is nothing but an encapsulated delegate.
- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the publisher class.
- Some other class that accepts this event is called the subscriber class. Events use the publisher-subscriber model.

C# – Events

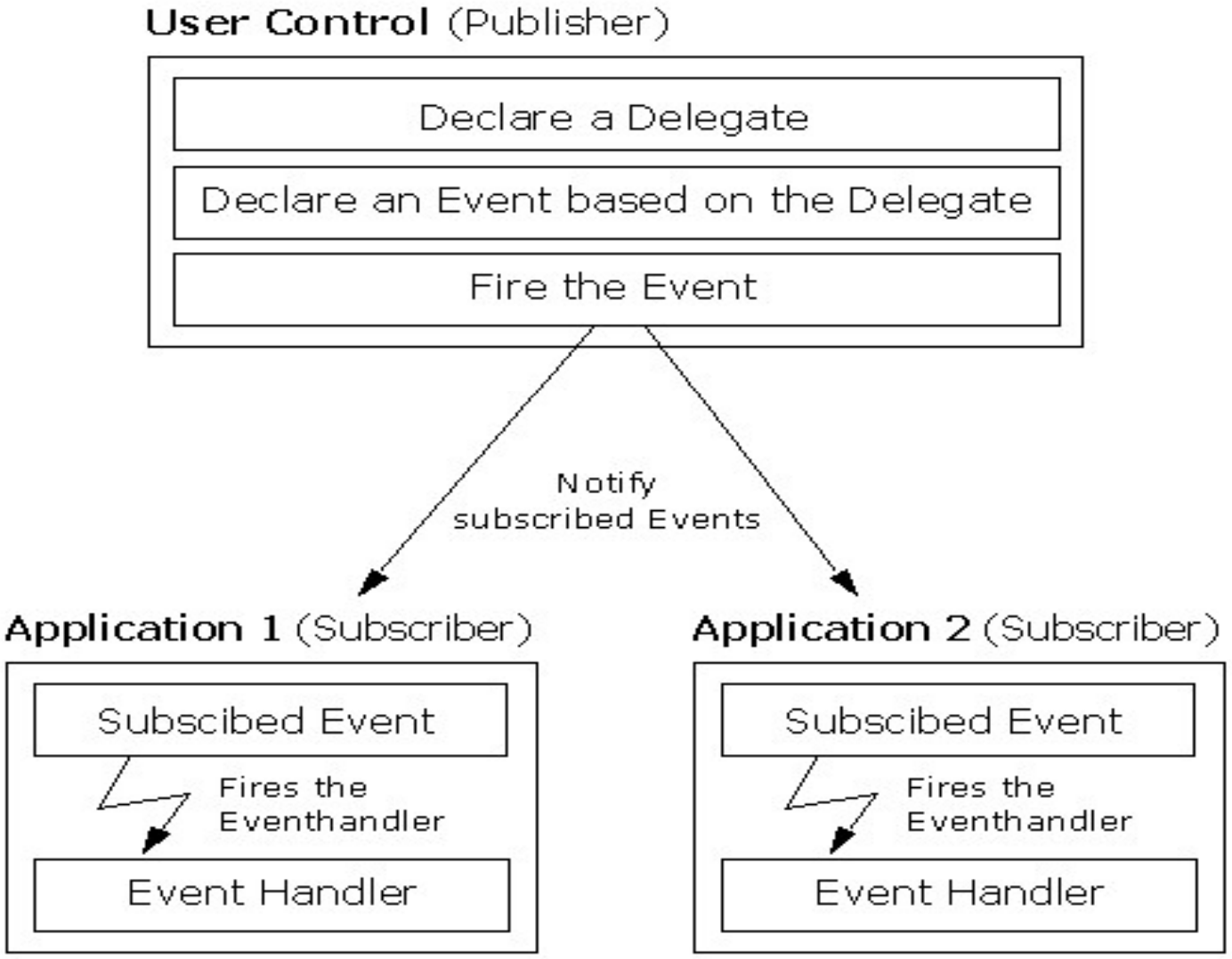


Even Handling in C# (A Publisher-Subscriber Model

C# – Events

- A publisher is an object that contains the definition of the event and the delegate.
- The event-delegate association is also defined in this object.
- A publisher class object invokes the event and it is notified to other objects.
- A subscriber is an object that accepts the event and provides an event handler.
- The delegate in the publisher class invokes the method (event handler) of the subscriber class.

C# - Events


User Control (Publisher)
Declare a Delegate
Declare an Event based on the Delegate
Fire the Event
Notify
subscribed Events
Application 1 (Subscriber)
Subscibed Event
Fires the
Eventhandler
Event Handler
Application 2 (Subscriber)
Subscibed Event
Fires the
Eventhandler
Event Handler

C# – Events

- **Declaring Events**
- To declare an event inside a class, first a delegate type for the event must be declared.
- For example,

  public delegate void BoilerLogHandler(string status);
- Next, the event is declared, using the **event** keyword:

```
//Defining event based on the above delegate
 public event BoilerLogHandler BoilerEventLog;
```

- The above code defines a delegate named *BoilerLogHandler* and an event named *BoilerEventLog*, which invokes the delegate when it is raised.

C# – Events

The following important conventions are used with events:

- The publishers determines when an event is raised and the subscriber determines what action is taken in response.
- An Event can have so many subscribers.
- Events are basically used for the single user action like button click.
- Event Handlers in the .NET Framework return void and take two parameters.
- The first parameter is the source of the event; that is the publishing object.
- The second parameter is an object derived from EventArgs.
- Events are properties of the class publishing the event.
- The keyword event controls how the event property is accessed by the subscribing classes.

Combining delegates

- Delegates can be combined such that when you call the delegate, a whole list of methods are called – potentially with different targets.
- Combined delegates can themselves be combined together, effectively creating one big list of simple delegates in the obvious fashion.
- Combining two delegate instances is usually done using the addition operator, as if the delegate instances were strings or numbers.
- Subtracting one from another is usually done with the subtraction operator.
- Note that when you subtract one combined delegate from another, the subtraction works in terms of lists.
- If the list to subtract is not found in the original list, the result is just the original list. Otherwise, the last occurrence of the list is removed

Combining delegates

- Delegate instances can also be combined with the static Delegate.
- Combined method, and one can be subtracted from another with the static Delegate.Remove method.
- The C# compiler converts the addition and subtraction operators into calls to these methods.
- Because they are static methods, they work easily with null references.

Expression	Result
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] - d1	d2
[d1, d2] - d2	d1
[d1, d2, d1] - d1	[d1, d2]
[d1, d2, d3] - [d1, d2]	d3
[d1, d2, d3] - [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] - [d1, d2]	[d1, d2, d3]
[d1, d2] - [d1, d2]	null

```
using System;
namespace Delegates {
   public delegate void DelEventHandler();

   class Program    {
      public static event DelEventHandler add;
      static void Main(string[] args)       {
         add += new DelEventHandler(USA);
         add += new DelEventHandler(India);
         add += new DelEventHandler(England);
         add.Invoke();
         Console.ReadLine();
      }
      static void USA()       {
         Console.WriteLine("USA");
      }
      static void India()       {
         Console.WriteLine("India");
      }
      static void England()       {
         Console.WriteLine("England");
      }
   }
}
```

C# – File I/O

- A file is a collection of data stored in a disk with a specific name and a directory path.
- When a file is opened for reading or writing, it becomes a stream.
- There are two main streams: the input stream and the output stream.
- The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation).
- The System.IO namespace has various classes used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.
- Following are some classes of this namespace.

Classes	Description
File	Helps in manipulating files.
FileStream	This class provides a Stream for a file, supporting both synchronous and asynchronous read and write operations.
StreamReader	This class implements a TextReader that reads characters from a byte stream in a particular encoding.
StreamWriter	This class implements a TextWriter for writing characters to a stream in a particular encoding.
TextReader	This class represents a reader that can read a sequential series of characters.
TextWriter	This class represents a writer that can write a sequential series of characters. This class is abstract.
BinaryReader	This class reads primitive data types as binary values in a specific encoding.
BinaryWriter	This class writes primitive types in binary to a stream and supports writing strings in a specific encoding.
StringReader	This class implements a TextReader that reads from a string.
StringWriter	This class implements a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.
FileInfo	This class provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. This class cannot be inherited.
DirectoryInfo	This class exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.

C# – File I/O

The FileStream Class

- The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.
- You need to create a **FileStream** object to create a new file or open an existing file.

- The **StreamReader** and **StreamWriter** class helps Reading from and Writing into Text files
- The **BinaryReader** and **BinaryWriter** class helps Reading from and Writing into Binary files
- C# allows you to work with the directories and files using various directory and file related classes like, the **File** class, **DirectoryInfo** class and the **FileInfo** class.

C# – File I/O

- You need to include namespace Sysmte.IO for using file handling classes
- **syntax for creating a FileStream object**
- FileStream <object_name> = new FileStream(<file_name>, <FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
- Example

  FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);

C# – File I/O

```
class Program     {
        public static void Main(string[] args)         {
            FileStream fi = new FileStream("f:\\file.txt",
             FileMode.OpenOrCreate);
              fi.WriteByte(12);
             fi.WriteByte(6);
              fi.WriteByte(30);
              fi.Close();
              FileStream fo = new FileStream("f:\\file.txt",
             FileMode.OpenOrCreate);
              int i = 0;
              Console.WriteLine("The contents of the file are:");
            while ((i = fo.ReadByte()) != -1)             {
                Console.WriteLine(i);
            }
            Console.ReadKey();
            fo.Close();
        }
    }
```

C# – File I/O

- **StreamReader and StreamWriter Class**
- The StreamReader Class reads the characters from a byte stream using a TextReader.
- The Read() and ReadLine() methods are used to read the data from the stream. The StreamWriter Class writes the characters to a byte stream.

C# - File I/O

```
class Program     {
        public static void Main(string[] args)        {
            FileStream f1 = new FileStream("e:\\file.txt",
             FileMode.OpenOrCreate);
            StreamWriter s1 = new StreamWriter(f1);
            s1.WriteLine("File Handling in C#");
            s1.Close();
            f1.Close();
            FileStream f2 = new FileStream("e:\\file.txt",
             FileMode.OpenOrCreate);
            StreamReader s2 = new StreamReader(f2);
            string data = s2.ReadLine();
            Console.WriteLine("The data in the file is as follows:");
            Console.WriteLine(data);
            Console.ReadKey();
            s2.Close();
            f2.Close();
        }
    }
```

C# – File I/O

- **TextReader Class and TextWriter Class**
- The TextReader Class has a reader that reads the characters from the file while the TextWriter Class has a writer that writes the characters into the file.

C# – File I/O

```
class Program    {
        static void Main(string[] args)        {
            using (TextWriter tw = File.CreateText("e:\\file.txt"))  {
                tw.WriteLine("C# File Handling");
                tw.WriteLine("TextReader Class and TextWriter
                      Class");
            }
            using (TextReader tr = File.OpenText("e:\\file.txt"))
            {
                Console.WriteLine(tr.ReadToEnd());
            }
        }
    }
```

C# – File I/O

Parameter	Description
FileMode	The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are: • **Append**: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist.
	• **Create**: It creates a new file. • **CreateNew**: It specifies to the operating system, that it should create a new file. • **Open**: It opens an existing file. • **OpenOrCreate**: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. • **Truncate**: It opens an existing file and truncates its size to zero bytes.

C# – File I/O

Parameter	Description
FileShare	**FileShare** enumerators have the following members: • **Inheritable**: It allows a file handle to pass inheritance to the child processes • **None**: It declines sharing of the current file • **Read**: It allows opening the file for reading • **ReadWrite**: It allows opening the file for reading and writing • **Write**: It allows opening the file for writing
FileAccess	**FileAccess** enumerators have members: **Read**, **ReadWrite** and **Write.**