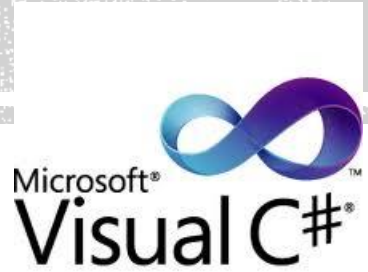
C# BASIC TUTORIAL (3)

GVC Ths. Nguyễn Minh Đạo

BM CNPM - Khoa CNTT - ĐHSPKT TP.HCM



OBJECTIVES

- C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg.
- This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.

C# - STRINGS

In C#, you can use strings as array of characters, however, more common practice is to use the **string** keyword to declare a string variable. The string keyword is an alias for the **System.String** class.

CREATING A STRING OBJECT

- You can create string object using one of the following methods:
  - 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

CREATING A STRING OBJECT

```
namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //from string literal and string concatenation
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //by using string constructor
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);

            //methods returning string
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //formatting method to convert a value
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
            string chat = String.Format("Message sent at {0:t} on {0:D}",
            waiting);
            Console.WriteLine("Message: {0}", chat);
            Console.ReadKey() ;
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

PROPERTIES OF THE STRING CLASS

- The String class has the following two properties:

S.N	Property Name & Description
1	**Chars** Gets the *Char* object at a specified position in the current *String* object.
2	**Length** Gets the number of characters in the current String object.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
1	**public static int Compare(string strA, string strB)** Compares two specified string objects and returns an integer that indicates their relative position in the sort order.
2	**public static int Compare(string strA, string strB, bool ignoreCase)** Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true.
3	**public static string Concat(string str0, string str1)** Concatenates two string objects.
4	**public static string Concat(string str0, string str1, string str2)** Concatenates three string objects.
5	**public static string Concat(string str0, string str1, string str2, string str3)** Concatenates four string objects.
6	**public bool Contains(string value)** Returns a value indicating whether the specified string object occurs within this string.
7	**public static string Copy(string str)** Creates a new String object with the same value as the specified string.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
8	**public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)** Copies a specified number of characters from a specified position of the string object to a specified position in an array of Unicode characters.
9	**public bool EndsWith(string value)** Determines whether the end of the string object matches the specified string.
10	**public bool Equals(string value)** Determines whether the current string object and the specified string object have the same value.
11	**public static bool Equals(string a, string b)** Determines whether two specified string objects have the same value.
12	**public static string Format(string format, Object arg0)** Replaces one or more format items in a specified string with the string representation of a specified object.
13	**public int IndexOf(char value)** Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	**public int IndexOf(string value)** Returns the zero-based index of the first occurrence of the specified string in this instance.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
15	**public int IndexOf(char value, int startIndex)** Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position.
16	**public int IndexOf(string value, int startIndex)** Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position.
17	**public int IndexOfAny(char[] anyOf)** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.
18	**public int IndexOfAny(char[] anyOf, int startIndex)** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position.
19	**public string Insert(int startIndex, string value)** Returns a new string in which a specified string is inserted at a specified index position in the current string object.
20	**public static bool IsNullOrEmpty(string value)** Indicates whether the specified string is null or an Empty string.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
21	**public static string Join(string separator, params string[] value)** Concatenates all the elements of a string array, using the specified separator between each element.
22	**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.
23	**public int LastIndexOf(char value)** Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object.
24	**public int LastIndexOf(string value)** Returns the zero-based index position of the last occurrence of a specified string within the current string object.
25	**public string Remove(int startIndex)** Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
26	**public string Remove(int startIndex, int count)** Removes the specified number of characters in the current string beginning at a specified position and returns the string.
27	**public string Replace(char oldChar, char newChar)** Replaces all occurrences of a specified Unicode character in the current string object with the specified Unicode character and returns the new string.
28	**public string Replace(string oldValue, string newValue)** Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string.
29	**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.
30	**public string[] Split(char[] separator, int count)** Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return.
31	**public bool StartsWith(string value)** Determines whether the beginning of this string instance matches the specified string.

METHODS OF THE STRING CLASS

S.N	Method Name & Description
32	**public char[] ToCharArray()** Returns a Unicode character array with all the characters in the current string object.
33	**public char[] ToCharArray(int startIndex, int length)** Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length.
34	**public string ToLower()** Returns a copy of this string converted to lowercase.
35	**public string ToUpper()** Returns a copy of this string converted to uppercase.
36	**public string Trim()** Removes all leading and trailing white-space characters from the current String object.

EXAMPLES: COMPARING STRINGS

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0)
         {
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         }
         else
         {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is test and This is text are not equal.
```

EXAMPLES: STRING CONTAINS STRING

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "This is test";
         if (str.Contains("test"))
         {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
The sequence 'test' was found.
```

EXAMPLES: GETTING A SUBSTRING

```
using System;

namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "Last night I dreamt of San Pedro";
            Console.WriteLine(str);
            string substr = str.Substring(23);
            Console.WriteLine(substr);
        }
        Console.ReadKey() ;
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
San Pedro
```

EXAMPLES: JOINING STRINGS

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

C# - STRUCTURES

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

OVERVIEW

- Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:
  - Title
  - Author
  - Subject
  - Book ID

DEFINING A STRUCTURE

- To define a structure, you must use the **struct** statement. The **struct** statement defines a new data type, with more than one member for your program.
- For example, here is the way you would declare the Book structure:

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

DEFINING A STRUCTURE

```
using System;

struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1;        /* Declare Book1 of type Book */
      Books Book2;        /* Declare Book2 of type Book */

      /* book 1 specification */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;

      /* book 2 specification */
      Book2.title = "Telecom Billing";
      Book2.author = "Zara Ali";
      Book2.subject =  "Telecom Billing Tutorial";
      Book2.book_id = 6495700;

      /* print Book1 info */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}", Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

      /* print Book2 info */
      Console.WriteLine("Book 2 title : {0}", Book2.title);
      Console.WriteLine("Book 2 author : {0}", Book2.author);
      Console.WriteLine("Book 2 subject : {0}", Book2.subject);
      Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

      Console.ReadKey();

   }
}
```

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

FEATURES OF C# STRUCTURES

- You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:
  - Structures can have methods, fields, indexers, properties, operator methods, and events.
  - Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and can't be changed.
  - Unlike classes, structures cannot inherit other structures or classes.

FEATURES OF C# STRUCTURES

- You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:
  - Structures cannot be used as a base for other structures or classes.
  - A structure can implement one or more interfaces.
  - Structure members cannot be specified as abstract, virtual, or protected.

FEATURES OF C# STRUCTURES

- You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:
  - When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
  - If the New operator is not used, the fields will remain unassigned and the object cannot be used until all the fields are initialized.

CLASS VS STRUCTURE

- Classes and Structures have the following basic differences:
  - classes are reference types and structs are value types
  - structures do not support inheritance
  - structures cannot have default constructor

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }
   public void display()
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};
```

```
public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1 = new Books(); /* Declare Book1 of type Book */
      Books Book2 = new Books(); /* Declare Book2 of type Book */

      /* book 1 specification */
      Book1.getValues("C Programming",
      "Nuha Ali", "C Programming Tutorial",6495407);

      /* book 2 specification */
      Book2.getValues("Telecom Billing",
      "Zara Ali", "Telecom Billing Tutorial", 6495700);

      /* print Book1 info */
      Book1.display();

      /* print Book2 info */
      Book2.display();

      Console.ReadKey();

   }
}
```

```
using System;

struct Books
{
   private string title;
   private string
   private string
   private int bo
   public void ge
   {
      title = t;
      author = a;
      subject = s
      book_id = i
   }
   public void di
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};
```

```
public class testStructure
{
   public static void Main(string[] args)
   {
                                                    Book1 of type Book */
                                                    Book2 of type Book */
                                                    6495407);
                                                    ", 6495700);
      /* print Book2 info */
      Book2.display();

      Console.ReadKey();

   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

C# - ENUMS

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

DECLARING *ENUM* VARIABLE

- The general syntax for declaring an enumeration is:

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

- Where,
  - The **enum_name** specifies the enumeration type name.
  - The enumeration list is a comma-separated list of identifiers.
- Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0.

DECLARING *ENUM* VARIABLE

The following example demonstrates use of enum variable:

```
using System;
namespace EnumApplication
{
   class EnumProgram
   {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args)
      {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
         Console.WriteLine("Friday: {0}", WeekdayEnd);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Monday: 1
Friday: 5
```

C# - CLASSES

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

CLASS DEFINITION

- A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces.
- Following is the general form of a class definition:

```
<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
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

```
<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
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

CLASS DEFINITION

- Please note that,
  - 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.

CLASS DEFINITION EXAMPLE

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // Length of a box
       public double breadth;  // Breadth of a box
       public double height;   // Height of a box
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // Declare Box1 of type Box
            Box Box2 = new Box();        // Declare Box2 of type Box
            double volume = 0.0;         // Store the volume of a box here

            // box 1 specification
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Volume of Box1 : {0}",  volume);

            // volume of box 2
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}", volume);
            Console.ReadKey();
        }
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

MEMBER FUNCTIONS AND ENCAPSULATION

- A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.
- Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

```
using System;
namespace BoxApplication
{
    class Box
    {
       private double length;   // Length of a box
       private double breadth;  // Breadth of a box
       private double height;   // Height of a box
       public void setLength( double len )
       {
            length = len;
       }

       public void setBreadth( double bre )
       {
            breadth = bre;
       }

       public void setHeight( double hei )
       {
            height = hei;
       }
       public double getVolume()
       {
           return length * breadth * height;
       }
    }
```

```
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // Declare Box1 of type Box
            Box Box2 = new Box();
            double volume;


            // Declare Box2 of type Box
            // box 1 specification
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // box 2 specification
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // volume of box 1
            volume = Box1.getVolume();
            Console.WriteLine("Volume of Box1 : {0}" ,volume);

            // volume of box 2
            volume = Box2.getVolume();
            Console.WriteLine("Volume of Box2 : {0}", volume);

            Console.ReadKey();
        }
    }
}
```

```
using System;
namespace BoxApplication
{
    class Box
    {
       private double length;   // Length of a box
       private double breadth;  // Breadth of a box
       private double height;   // Height of a box
       public void setLength( double len )
       {
           length = len;
       }
       public void setBreadth( double bre )
       {
           breadth = bre;
       }
       public void setHeight( double hei )
       {
            height = hei;
       }
       public double getVolume()
       {
           return length * breadth * height;
       }
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // Declare Box1 of type Box
            Box Box2 = new Box();
            double volume;


            // Declare Box2 of type Box
            // box 1 specification
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // box 2 specification
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // volume of box 1
            volume = Box1.getVolume();
            Console.WriteLine("Volume of Box1 : {0}" ,volume);

            // volume of box 2
            volume = Box2.getVolume();
            Console.WriteLine("Volume of Box2 : {0}", volume);

            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

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

CONSTRUCTORS IN C#

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()
      {
         Console.WriteLine("Object is being created");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

CONSTRUCTORS IN C#

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()
      {
         Console.WriteLine("Object is being created");
      }

      public void setLength( double len )
      {
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

```
      static void Main(string[] args)
      {
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

DEFAULT CONSTRUCTOR

- A **default constructor** does not have any parameter but if you need a constructor can have parameters.
- Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation.

DEFAULT CONSTRUCTOR

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line(double len)  //Parameterized constructor
      {
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

DEFAULT CONSTRUCTOR

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line(double len)  //Parameterized constructor
      {
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

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.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc. Destructors cannot be inherited or overloaded.

DESTRUCTORS IN C#

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()  // constructor
      {
         Console.WriteLine("Object is being created");
      }
      ~Line() //destructor
      {
         Console.WriteLine("Object is being deleted");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }
```

```
      static void Main(string[] args)
      {
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

DESTRUCTORS IN C#

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()  // constructor
      {
         Console.WriteLine("Object is being created");
      }
      ~Line() //destructor
      {
         Console.WriteLine("Object is being deleted");
      }

      public void setLe
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

```
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

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

STATIC MEMBERS OF A C# CLASS

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;
        public void count()
        {
            n
        }
        publi
        {
            r
        }
    }
```

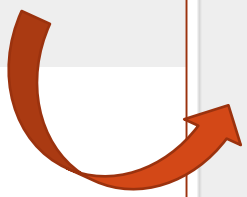
When the above code is compiled and executed, it produces the following result:

```
Variable num for s1: 6
Variable num for s2: 6
```

```
    class StaticTester
                s1.count();
                s2.count();
                s2.count();
                s2.count();
                Console.WriteLine("Variable num for s1: {0}", s1.getNum());
                Console.WriteLine("Variable num for s2: {0}", s2.getNum());
                Console.ReadKey();
            }
        }
    }
```

STATIC MEMBERS OF A C# CLASS

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;
        public void count()
        {
            num++;
        }
        public int getNum()
        {
            return num;
        }
    }
```



```
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();
            s1.count();
            s1.count();
            s1.count();
            s2.count();
            s2.count();
            s2.count();
            Console.WriteLine("Variable num for s1: {0}", s1.getNum());
            Console.WriteLine("Variable num for s2: {0}", s2.getNum());
            Console.ReadKey();
        }
    }
}
```

STATIC MEMBERS OF A C# CLASS

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s = new StaticVar();
            s.count();
            s.count();
            s.count();
            Console.WriteLine("Variable num: {0}", StaticVar.getNum());
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Variable num: 3
```

C# - INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

OVERVIEW

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well and so on.

BASE AND DERIVED CLASSES

- **A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base class or interface.**
- The syntax used in C# for creating derived classes is as follows:

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

BASE AND DERIVED CLASSES

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }
```

```
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

BASE AND DERIVED CLASSES

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      publi
      {
         he
      }
      prote
      prote
   }

   // Derived class
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

```
            Rectangle Rect = new Rectangle();

            Rect.setWidth(5);
            Rect.setHeight(7);

            // Print the area of the object.
            Console.WriteLine("Total area: {0}",  Rect.getArea());
            Console.ReadKey();
        }
    }
}
```

BASE CLASS INITIALIZATION

- The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created.
- You can give instructions for superclass initialization in the member initialization list.

BASE CLASS INITIALIZATION

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
```

```
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
         base.Display();
         Console.WriteLine("Cost: {0}", GetCost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

BASE CLASS INITIALIZATION

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w
      }
      public doubl
      {
         return le
      }
      public void
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
```

```
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
                                                    ost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

MULTIPLE INHERITANCE IN C#

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

MULTIPLE INHERITANCE IN C#

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Base class PaintCost
   public interface PaintCost
   {
      int getCost(int area);

   }
```

```
   // Derived class
   class Rectangle : Shape, PaintCost
   {
      public int getArea()
      {
         return (width * height);
      }
      public int getCost(int area)
      {
         return area * 70;
      }
   }
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();
         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

MULTIPLE INHERITANCE IN C#

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
      }
      protec
      protec
   }

   // Base c
   public interface PaintCost
   {
      int getCost(int area);

   }
```

```
   // Derived class
   class Rectangle : Shape, PaintCost
   {
      public int getArea()
      {
         return (width * height);
      }
      public int getCost(int area)
      {
         return area * 70;
      }
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
Total paint cost: $2450
```

```
      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();
      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
      Console.ReadKey();
   }
}
}
```

C# - POLYMORPHISM

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

OVERVIEW

- The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.
- Polymorphism can be static or dynamic. 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

- The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding.
- C# provides two techniques to implement static polymorphism. These are:
  - Function overloading
  - Operator overloading

FUNCTION OVERLOADING

- You can have multiple definitions for the same function name in the same scope.
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
- You cannot overload function declarations that differ only by return type.

FUNCTION OVERLOADING

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s)
      {
         Console.WriteLine("Printing string: {0}", s);
      }
```

```
      static void Main(string[] args)
      {
         Printdata p = new Printdata();
         // Call print to print integer
         p.print(5);
         // Call print to print float
         p.print(500.263);
         // Call print to print string
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

FUNCTION OVERLOADING

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.
      }

      void print(
      {
         Console.
      }
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

```
args)

tdata();
integer

float
            p.print(500.263);
            // Call print to print string
            p.print("Hello C++");
            Console.ReadKey();
         }
      }
   }
```

DYNAMIC POLYMORPHISM

- C# allows you to create **abstract classes** that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it.
- **Abstract classes** contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

DYNAMIC POLYMORPHISM

- Please note the following rules about **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.

ABSTRACT CLASSES

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle class area :");
         return (width * length);
      }
   }
```

```
class RectangleTester
{
   static void Main(string[] args)
   {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
   }
}
}
```

ABSTRACT CLASSES

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length =
         width =
      }
      public ove
      {
         Console
         return (width * length);
      }
   }
```

```
class RectangleTester
                                                    s)
                                                    e(10, 7);
                Console.WriteLine("Area: {0}",a);
                Console.ReadKey();
            }
        }
    }
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area :
Area: 70
```

DYNAMIC POLYMORPHISM

- 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.
- Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

DYNAMIC POLYMORPHISM

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("Parent class area :");
         return 0;
      }
   }
```

```
class Rectangle: Shape
{
   public Rectangle( int a=0, int b=0): base(a, b)
   {

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

```
class Triangle: Shape
{
   public Triangle(int a = 0, int b = 0): base(a, b)
   {

   }
   public override int area()
   {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
   }
}
```

DYNAMIC POLYMORPHISM

```
    class Caller
    {
        public void CallArea(Shape sh)
        {
            int a;
            a = sh.area();
            Console.WriteLine("Area: {0}", a);
        }
    }
    class Tester
    {

        static void Main(string[] args)
        {
            Caller c = new Caller();
            Rectangle r = new Rectangle(10, 7);
            Triangle t = new Triangle(10, 5);
            c.CallArea(r);
            c.CallArea(t);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```


Q
A


Thank
you!