C# BASIC TUTORIAL (4)

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# - INTERFACES

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the '**what**' part of the syntactical contract and the deriving classes define the '**how**'part of the syntactical contract.

OVERVIEW

- Interfaces define properties, methods and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.
- Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

DECLARING INTERFACES

- Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

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

EXAMPLE

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

namespace InterfaceApplication
{

   public interface ITransactions
   {
      // interface members
      void showTransaction();
      double getAmount();
   }
   public class Transaction : ITransactions
   {
      private string tCode;
      private string date;
      private double amount;
      public Transaction()
      {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }
      public Transaction(string c, string d, double a)
      {
         tCode = c;
         date = d;
         amount = a;
      }
      public double getAmount()
      {
         return amount;
      }
      public void showTransaction()
      {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());

      }

   }
   class Tester
   {
      static void Main(string[] args)
      {
         Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
         t1.showTransaction();
         t2.showTransaction();
         Console.ReadKey();
      }
   }
}
```

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

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

C# - NAMESPACES

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace will not conflict with the same class names declared in another.

DEFINING A NAMESPACE

- A namespace definition begins with the keyword namespace followed by the namespace name as follows:

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

- To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name.item_name;
```

EXAMPLE

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
```

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

```
Inside first_space
Inside second_space
```

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

THE *USING* KEYWORD

- The using keyword states that the program is using the names in the given namespace. For example, we are using the System namespace in our programs. The class Console is defined there. We just write:

  Console.WriteLine ("Hello there");

- We could have written the fully qualified name as:

  System.Console.WriteLine("Hello there");

- You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

EXAMPLE

```
using System;
using first_space;
using second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class efg
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
```

```
class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

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

```
Inside first_space
Inside second_space
```

NESTED NAMESPACES

- Namespaces can be nested where you can define one namespace inside another namespace as follows:

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

- You can access members of nested namespace by using the dot (.) operator

EXAMPLE

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
   namespace second_space
   {
      class efg
      {
         public void func()
         {
            Console.WriteLine("Inside second_space");
         }
      }
   }
}
```

```
class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

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

```
Inside first_space
Inside second_space
```

C# - PREPROCESSOR DIRECTIVES

The preprocessors directives give instruction to the compiler to preprocess the information before actual compilation starts.

OVERVIEW

- All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).
- C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation. Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

LIST OF PREPROCESSOR DIRECTIVES IN C#

Preprocessor Directive	Description.
#define	It defines a sequence of characters, called symbol.
#undef	It allows you to undefine a symbol.
#if	It allows testing a symbol or symbols to see if they evaluate to true.
#else	It allows to create a compound conditional directive, along with #if.
#elif	It allows creating a compound conditional directive.
#endif	Specifies the end of a conditional directive.
#line	It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.
#error	It allows generating an error from a specific location in your code.
#warning	It allows generating a level one warning from a specific location in your code.
#region	It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.
#endregion	It marks the end of a #region block.

THE #DEFINE PREPROCESSOR

- The **#define** preprocessor directive creates symbolic constants.
- **#define** lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to true. Its syntax is as follows:

  #define symbol

```
#define PI
using System;
namespace PreprocessorDAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

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

```
PI is defined
```

CONDITIONAL DIRECTIVES

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
        }
    }
```

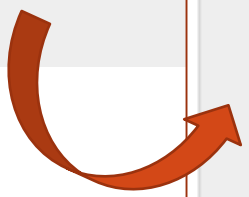
When the above code is compiled and executed, it produces the following result:

```
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!