Language Preliminaries Contd.

Structs

A struct is a user-defined type that represents data structures. It is similar to a class, with the following key differences:

- A struct is a value type, where a class is a reference type.
- A struct does not support inheritance

A struct can have all of the members that a class can, except a parameterless constructor, field initializers, a finalizer, virtual or protected members. In C#, structures are defined using the struct keyword. The default modifier is internal for the struct and its members.

The general structure for defining a struct is as shown below:

```
struct StructureName
{
     //data members
}
```

A struct object can be created with or without the new operator, same as primitive type variables.

```
struct Point
{
      public int X;
      public int Y;
 }


Point pt;
pt.X = 100;
pt.Y = 50;
```

A struct can contain properties, auto-implemented properties, methods, etc., same as classes.

```
struct Point
{
   public int X { get; set; }
   public int Y { get; set; }

  public void Increment()
```

```
        {
            X++; Y++;
        }
    }
```

A struct cannot contain a parameterless constructor. It can only contain parameterized constructors or a static constructor. When you define a struct constructor, you must explicitly assign every field.

```
struct Point
 {
        public int x;
        public int y;
        public  Point(int x, int y)
        {
            this.x = x; this.y = y;
        }


}

Point p1 = new Point();
Point p2 = new Point(2, 4);
```

The **readonly** modifier on a struct definition declares that the struct is immutable. The Every instance field of the struct must be marked readonly, as shown in the following example:

```
readonly struct Point
{
     public readonly int X, Y; // X and Y must be readonly
}
```

It guarantees that no member of the struct can manipulate its content as it ensures every field is marked as readonly. Adding a field not marked as readonly will generate the following compiler error:

CS8340: "Instance fields of readonly structs must be readonly."

This guarantee is important because it allows the compiler to avoid defensive copies of struct values.

Unlike reference types, whose instances always live on the heap, value types live in-place wherever the variable was declared. If a value type appears as a parameter or local variable, it will reside on the stack:

```
struct Point { public int X, Y; }
void SomeMethod()
{
     Point pt; // pt will reside on the stack
}
```

Ref Structs: A ref struct is basically a structure that can only live on the execution stack.

Sample Code: `ref struct Point { }`

Attempting to use a ref struct in such a way that it could reside on the heap generates a compile-time error:

```
ref struct Point
{
      public int X, Y;
}
class SampleClass
{
      Point pt;
 }
```

The sample code above does not compile since this would automatically place the ref struct onto the heap.

Enums

An enum type is a special data type that enables for a variable to be a set of predefined constants.The enum is defined using the enum keyword, directly inside a namespace, class, or structure. For example:

```
public enum Direction
{
     NORTH,
     SOUTH,
     EAST,
     WEST
};
```

Each enum member has an underlying integral value. These are, by default:

- Underlying values are of type int.
- The constants 0, 1, 2... are automatically assigned, in the declaration order of the enum members.

You can also specify an explicit underlying value for each enum member. For example:

```
enum Status
{
   START = 1,
   STOP = 0
};
```

You can assign different values to enum member. A change in the default value of an enum member will automatically assign incremental values to the other members sequentially.

```
public enum Direction
{
     NORTH,    //0
     SOUTH,    //1
     EAST = 3, //3
     WEST      //4
};
```

This enum can be used as follows:

```
     Direction value = Direction.WEST;
     Console.WriteLine(value); //output: WEST
```

Explicit casting is required to convert from an enum type to its underlying integral type.

```
 int value  = (int) Direction.WEST;
 Console.WriteLine(value);
//output: 4
```

Benefits of using Enum

- Enum provides type-safe; Enum variable can be assigned only with predefined enum constants otherwise it will throw compilation error.
- Enum improves code clarity and makes program easier to maintain.
- Enum provides efficient way to assign multiple constant integral values to a single variable.

Abstract Class

A class that contains one or more abstract methods is called abstract class. An abstract class can never be instantiated. Instead, only its concrete subclasses can be instantiated. Abstract classes are able to define abstract members. Abstract members are like virtual members except that they don't provide a default implementation. That implementation must be provided by the subclass unless that subclass is also declared abstract. Also the abstract modifier can be used with indexers, events and properties.

The abstract modifier indicates the incomplete implementation. The keyword abstract is used before the class or method to declare the class or method as abstract.

```
abstract class Shape   //abstract class
{
    abstract public int CalculateArea(); //  abstract method
}
```

Abstract Method: A method which is declared abstract has no body and declared inside the abstract class only.

Sample Code:

```
abstract class Shape
{
    public abstract void Draw(); //Method with no body

}

class  Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Rectangle");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Circle");
    }
}

public class Program
```

```
    {
        static void Main(string[] args)
        {

            Shape shape = new Rectangle();
            shape.Draw();
            shape = new Circle();
            shape.Draw();

            Console.ReadLine();
        }
    }
```

In the example below, the class MyDerivedC is derived from an abstract class MouseCordinate. The abstract class contains an abstract method, MyMethod(), and two abstract properties, GetX() and GetY().

```
abstract class Mouse // Abstract class
{
      protected int x = 100;
      protected int y = 150;

      public abstract void ChangePosition(); // Abstract method
      public abstract int GetX // Abstract property
      {
            get;
      }
      public abstract int GetY // Abstract property
      {
            get;
      }
}
class MouseCordinate : Mouse
{
      public override void ChangePosition()
      {
            x++;
            y++;
      }
      public override int GetX // overriding property
      {
            get
            {
                  return x + 10;
             }
            }
            public override int GetY // overriding property
            {
                get
                {
                    return y + 10;
                }
            }
        }

        public class Program
        {
            public static void Main()
            {
                MouseCordinate mouseCordinate = new MouseCordinate();
                mouseCordinate.MyMethod();
                Console.WriteLine("x = {0}, y = {1}",
                mouseCordinate.GetX, mouseCordinate.GetY);
            }
        }
```

Sealed Class

A class which is marked with sealed keyword prevents inheritance from occurring. A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. When you mark a class as sealed ,the compiler will not allow you to derive from this type.

```
        public sealed class ClassName
        {
            // Class members here.
        }
```

In c#, a sealed class can be defined by using a sealed keyword. The sealed keyword tells the compiler that the class is sealed, and therefore, cannot be extended.

```
        public sealed class Configuration
        {
            // Class members here.
        }

        public class Detail : Configuration // Compilation fails
        {
        }
```

The sealed keyword can be used on a **method**, indexer, property, or event, on a derived class that overrides a virtual method or property in a base class to allow other classes to derive from base class but to prevent them from overriding specific virtual methods or properties.

```
class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Base Class");
    }
}

class SubClass : BaseClass
{
    public sealed override void Print()
    {
        Console.WriteLine("Sub Class");
    }
}

class DerivedClass : SubClass
{
  // cannot override inherited member because it is sealed
     public override void Print()
    {
        Console.WriteLine("Derived Class");
    }
}
```

In the example above a sealed keyword is used on method Print that overrides a base class virtual method, derived class cannot override the virtual Print method that is decleared in Base class because that method is sealed in SubClass.

Interfaces

An interface is a kind of type that is used to specify a behavior that classes must implement. They are similar to protocols. Interfaces are declared using the interface keyword, and may only contain method signature.

```
public interface Example
{
    void show();
}
```

An interface can contain **declarations** of methods, properties, indexers, and events. However, it cannot contain fields, auto-implemented properties.

- An interface is like an abstract class, which can not be used to create objects.
- An interface cannot include private, protected, or internal members.
- Any class implementing the interface must implement all of its members.
- Interface can contain signatures for methods, properties, events.
- Interface contain no implementation of methods.
- A class or struct can implement multiple interfaces. In contrast, a class can inherit from only a single class, and a struct cannot inherit at all.
- An interface itself can inherit from multiple interfaces.
- Interface can not contain constants, fields, instance constructors, destructors or types.
- By default, all the members of an interface are public and abstract.
- Interfaces provide a way to achieve runtime polymorphism.

Interface Implementation

```
interface VehicleInterface
 {
     void Start();
     void Stop();
 }
 class Vehicle : VehicleInterface
 {
     public void Start()
     {
         Console.WriteLine("Vehicle Started");
     }

     public void Stop()
     {
         Console.WriteLine("Vehicle Stopped");
     }
 }
```

The above sample code demonstrates interface implementation. In which, the interface VehicleInterface contains two method declarations which are resposible for starting and stopping the vehicle.

Implementing Multiple Interfaces: A class or struct can implement multiple interfaces. It must provide the implementation of all the members of all interfaces**.**

```
class Employee : EmployeeInterface, IpersonInterface
{
     //must implement all the members of both the interfaces
}
```

Delegate and Events

A delegate is a type-safe object that points to another method. Delegate is a reference type, however, instead of referring to an object, it refers to a method. It is often used to implement callbacks and event handlers.

The declaration of delegate will be same as method signature but the only difference is delegate keyword is used to define delegates.Speficically, it defines the method's return type and its parameter types.
The syntax of defining a delegate using delegate keyword:

```
    public delegate void SampleDelegate(int a, int b);
```

Instantiating a delegate:

```
    SampleDelegate sampleDelegate = SampleMethod;
```

Also, new keyword can be used to instantiate a delegate.

```
    SampleDelegate sampleDelegate = new SampleDelegate(SampleMethod);
```

Sample Code:

```
        public delegate int MathDelegate(int a, int b);
        public class Program
        {
            public static int Add(int a, int b)
            {
                return a + b;
            }
            public static int Subtract(int a, int b)
            {
                return a - b;
            }
            static void Main(string[] args)
            {
                MathDelegate mathDelegate = Add;
                var result = mathDelegate(10, 20);
                Console.WriteLine(result);

                mathDelegate = Subtract;
                result = mathDelegate(10, 20);
                Console.WriteLine(result);

                Console.ReadLine();
            }
        }
```

Delegates are similar to the function pointers in C++ but are type safe.Delegates allow methods to pass as parameters.

Multicast Delegate

Multicast delegate is a delegate which holds a reference to more than one method. All delegate instances have multicast capability. Multicast delegates must contain only methods that return void, else it will return value of the last method method on the list.

The `+` and `+=` operators add the multiple method references to the delegate object. Delegates are invoked in the order in which they are added.

```
SampleDelegate sampleDelegate = SampleMethod;
sampleDelegate += AnotherMethod;
```

```
Sample Code:
    delegate void MathDelegate(int a, int b);
    public class Program
    {
        static void Add(int a, int b)
        {
            Console.WriteLine("A + B : {0}", a + b);
        }
        static void Subtract(int a, int b)
        {
            Console.WriteLine("A - B : {0}", a - b);
        }
        static void Main(string[] args)
        {
            MathDelegate mathDelegate = new MathDelegate(Add);
            mathDelegate += Subtract;
            mathDelegate(10, 20);

            Console.ReadLine();
        }
    }
```

The `-` and `-=` operators remove the method refrences from the delegate object.

```
mathDelegate -= Add;
```

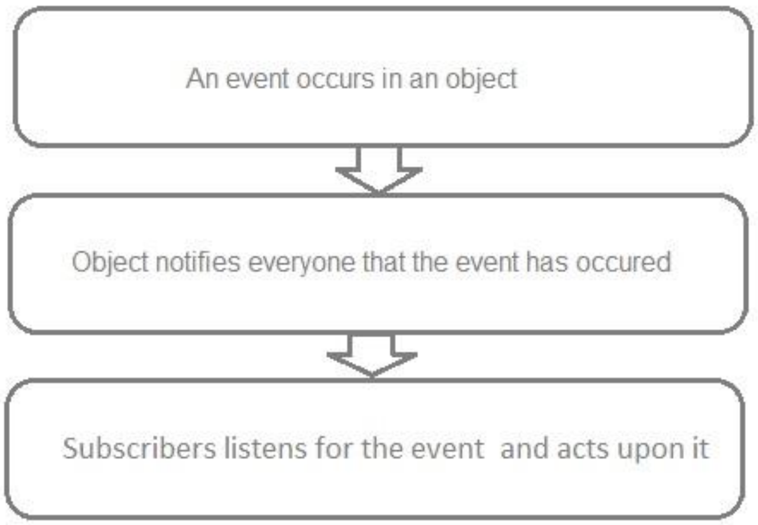
Events

An event is a mechanism for a class or object to provide notifcation to other classes or objects on the occurrence of an action. The action could be caused either by a button click, mouse movements or by some other programming logic. The object that sends or raises the event is called the publisher and the classes that receive or handle the event are called subscribers.

- Publisher determines when to raise an event; the subscriber determines what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple subscribers.
- Events that have no subscribers are never raised.

Event handling :

The following figure is generalized representation of events and event handling.


An event occurs in an object
Object notifies everyone that the event has occured
Subscribers listens for the event and acts upon it

An Event is associated with Event Handler using Delegates. When an Event raised, it sends a signal to delegates and delegates executes the right method. The following steps describe event handling process.

Step 1: Define a delegate.

To declare an event, first a delegate type of the event must be declared, if none is already declared.

```
public delegate void ChangedEventHandler(object sender, Eventargs e);
```

Step 2: Define an event with same name of delegate.

```
public event ChangedEventHandler Changed;
```

An event is declared like a field of delegate type, except that the event keyword precedes the event declaration, following the modifiers. Events usually are declared public, but any accessibility modifiers are allowed.

Step 3: Invoking an event.

Once a class has declared an event, it can treat that event just like a field of the indicated delegate type.The field will either be null, if no client has hooked up a delegate to the event, or else it refers to a delegate that should be called when the event is invoked. This invoking an event is generally done by first checking for null and then calling the event.

```
If (Changed != null)
{
     Changed(this,e);
 }
```

Invoking an event can only be done from within the class that declares the event.

Step 4 : Hooking up to an event.

This is done with the += and - = operators. To begin receive event notifications; client code first creates a delegate of the event type that refers to the method that should be invoked from the event. Then it composes that onto any other delegate that the event might be connected to using += operator.

```
List.Changed += new ChangedEventHandler(ListChanged);
```

When the client code is done receiving event notifications, it removes its delegate from the event by using - = operator.

Sample Code:

```
class MathOperation
{
    public delegate void OddNumberEventHandler();
    //Declared Delegate
    public event OddNumberEventHandler OddNumberEvent;
    //Declared Event

    public void Add()
    {
        int result;
```

```
            result = 2 + 3;
            Console.WriteLine("Addition : {0}", result);
            //Check if result is odd number then raise event
            if ((result % 2 != 0) && (OddNumberEvent != null))
            {
                OddNumberEvent(); //Raised Event
            }
        }
    }

    public class Program
    {

        static void Main(string[] args)
        {
            MathOperation mOperation = new MathOperation();
            //Event gets binded with delegate
            mOperation.OddNumberEvent +=
            new MathOperation.OddNumberEventHandler(EventMessage);
            mOperation.Add();
            Console.ReadLine();
        }
        static void EventMessage()
        {
            Console.WriteLine("Event Executed :
                                    This is a Odd   Number");
        }
    }
```

Output:

```
     Addition : 5
     Event Executed : This is a Odd Number
```

Sample Code (Complex Example):

```
using System;
using System.Collections;
namespace Demo
{
	public class Program
	{
		public static void Main()
		{
			// Create a new list.
			ListWithChangedEvent list = new 
ListWithChangedEvent();

			// Create a class that listens to the list's change 
event.
			EventListener listener = new EventListener(list);

			// Add and remove items from the list.
			list.Add("item 1");
			list.Clear();

			// detach the listener
			listener.Detach();
			Console.ReadLine();
		}
	}

	// A delegate type for hooking up change notifications.
	public delegate void ChangedEventHandler(object sender, EventArgs 
e);


	// A class that works just like ArrayList, but sends event
	// notifications whenever the list changes.
	public class ListWithChangedEvent : ArrayList
	{
		// An event that clients can use to be notified whenever the
		// elements of the list change.
		public event ChangedEventHandler Changed;


		// Invoke the Changed event; called whenever list changes
		protected virtual void OnChanged(EventArgs e)
		{
			if (Changed != null)
```

```
                    Changed(this, e);
            }

            // Override some of the methods that can change the list;
            // invoke event after each
            public override int Add(object value)
            {
                  int i = base.Add(value);
                  OnChanged(EventArgs.Empty);
                  return i;
            }

            public override void Clear()
            {
                  base.Clear();
                  OnChanged(EventArgs.Empty);
            }

            public override object this[int index]
            {
                  set
                  {
                        base[index] = value;
                        OnChanged(EventArgs.Empty);
                  }
            }
      }

      class EventListener
      {
            private ListWithChangedEvent List;

            public EventListener(ListWithChangedEvent list)
            {
                  List = list;
                  // Add "ListChanged" to the Changed event on "List".
                  List.Changed += new ChangedEventHandler(ListChanged);
            }

            // This will be called whenever the list changes.
            private void ListChanged(object sender, EventArgs e)
            {
                  Console.WriteLine("This is called when the event 
fires.");
            }
```

```
        public void Detach()
        {
            // Detach the event and delete the list
            List.Changed -= new ChangedEventHandler(ListChanged);
            List = null;
        }
        }
    }
```

Partial Class

It is possible to split the definition of a class, a struct, an interface or a method into multiple files. Each source file contains a section of the class or method definition, and all parts are combined when the application is compiled.

To split a class definition, use the partial keyword modifier, as shown below:

```
    partial class Student
    {
        public void DoHomeWork()
        {
        }
    }

    partial class Student
    {
        public void GoToLunch()
        {
        }
    }
```

The partial keyword is used to specify that other parts of the class, struct, or interface can be defined in the namespace into multiple files. Any class, struct, or interface members declared in a partial definition are available to all the other parts.
Each part must have the partial declaration; the following is not allowed:

```
    partial class Student
    {
        // ...
    }
```

```
class Student
{
    //...
}
```

Rules to Implement Partial Class

- All the partial class definitions must be in the same assembly and namespace.
- The partial modifier can only appear immediately before the keywords class, struct or interface.
- All the parts must have the same accessibility, such as public, private, and so on.
- All the parts must be available at compile time to form the final type.
- If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed.
- Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations.

Sample Code:

```
namespace Demo
{
    partial class Student
    {
        private string name;
        private string address;
        public Student(string _name, string _address)
        {
            this.name = _name;
            this.address = _address;
        }
    }
    partial class Student
    {
        public void PrintStudentDetail()
        {
            Console.WriteLine("Student Deails");
            Console.WriteLine("Name: {0}", name);
            Console.WriteLine("Address: {0}", address);
        }
    }
    public class Program
    {
        static void Main(string[] args)
        {
            Student std = new Student("Rajesh", "Kathmandu");
```

```
                std.PrintStudentDetail();
                Console.ReadLine();
            }
        }
    }
```

The following example shows how to develop partial structs and interfaces.

```
    partial interface ITest
    {
        void Interface_Method1();
    }

    partial interface ITest
    {
        void Interface_Method2();
    }

    partial struct S1
    {
        void Struct_Method1() { }
    }

    partial struct S1
    {
        void Struct_Method2() { }
    }
```

Partial Methods
A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

```
    // Definition in file1.cs
    partial void CalculateArea();

    // Implementation in file2.cs
    partial void CalculateArea()
    {
      // method body
    }
```

Key properties of partial methods

- The signature of partial method must be same in both partial classes.
- The partial method must return void.
- Partial methods can have in or ref but not out parameters.
- Partial methods are implicitly private, therefore they can not be virtual.

Collections

Collections are the enumerable data structures that can be accessed using indexes or keys. Closely related data can be handled more efficiently when grouped together into a collection. Instead of writing separate code to handle each individual object. We can use the same code to process all the elements of a collection.

To manage a collection, the array class and the System.Collections classes are used to add, remove and modify either individual elements of the collection or a range of elements.

There are 3 ways to work with collections which are follow.

System.Collections.Generic classes
System.Collections.Concurrent classes
System.Collections classes

The collection classes provide support for stacks, queues, lists and hash tables. Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs.

Sample Code:

```
        using System.Collections;
       // must import System.Collections to access the ArrayList
          public class Program
          {
              static void Main(string[] args)
              {
                  ArrayList strArray = new ArrayList();
                  strArray.Add("First"); //Add item
                  strArray.Add("Second");
                  strArray.Add("Third");
                  // Show number of items in ArrayList.
                 Console.WriteLine("This collection has {0} items.",
strArray.Count);
                  // Display contents.
                  foreach (string s in strArray)
                  {
                      Console.WriteLine("Entry: {0}", s);
                  }
                  Console.WriteLine();
                  // Remove second item and display current count.
                  strArray.Remove("Second");
                  Console.WriteLine("This collection has {0} items.",
      strArray.Count);
                  // Display contents.
                  foreach (string s in strArray)
                  {
                      Console.WriteLine("Entry: {0}", s);
                  }

                  Console.ReadLine();
              }
          }
```

In a collection, you don't need to define the size of the collection beforehand. You can add elements or even remove elements from the collection at any point of time.

Any reference or value type that is added to an ArrayList is implicitly upcast to Object. If the items are value types, they must be boxed when added to the list, and unboxed when they are retrieved. Both the casting and the boxing and unboxing operations degrade performance; the effect of boxing and unboxing can be quite significant in scenarios where you must iterate over large collections.
Most collections derive from the interfaces ICollection, IComparer, IEnumerable, IList, IDictionary and IDictionaryEnumerator and their generic equivalents.

Types of System.Collections

ArrayList:	The ArrayList collection is similar to the Arrays data type in C#. The biggest difference is the dynamic nature of the array list collection.
Stack:	The stack is a special case collection which represents a last in first out (LIFO) concept.
Queues:	The Queue is a special case collection which represents a first in first out concept.
Hashtable:	A hash table is a special collection that is used to store key-value items.
SortedList:	The SortedList is a collection which stores key-value pairs in the ascending order of key by default.
BitArray:	A bit array is an array of data structure which stores bits.

Sample Code for Stack:

```
                 Stack st = new Stack();
                 st.Push("First"); //Adds item
                 st.Push("Second");
                 st.Push("Third");

                 st.Pop(); //Removes item

                 foreach (Object obj in st)
                 {
                     Console.WriteLine(obj);
                 }
      Output:
                Second
                First
```

Sample Code for Queue:

```
                 Queue qt = new Queue();
                 qt.Enqueue("First"); //Adds Item
                 qt.Enqueue("Second");
                 qt.Enqueue("Third");

                 qt.Dequeue(); // Removes Item

                 foreach (Object obj in qt)
                 {
                     Console.WriteLine(obj);
                 }
      Output:
                Second
                Third
```

Generics

Generics is a feature or concept which enables the types to be more safely reusable and more efficient. It allows us to define classes and methods with placeholders, the compiler replaces these placeholders with specified type at compile time. Generics are most commonly used with collections and the methods that operate on them. The concept of generics is used to create general purpose classes and methods.
The System.Collections.Generic namespace contains several generic-based collection classes.
List<T> equivalent of the ArrayList, which implements IList<T>. It comes under System.Collection.Generic namespace. So, you must import System.Collections.Generic to access the List.

An implementation of List<T> generic collection.

```
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
List<string> stringList = new List<string>();
stringList.Add("abc");
stringList.Add("xyz");
```

An implementation of Queue<T> generic collection.

```
Queue<string> stringQ = new Queue<string>();
stringQ.Enqueue("abc");
stringQ.Enqueue("xyz");
```

you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient.
Sample Code:

```
//Declare the generic class
public class GenericClass<T>
{
    public void GenericFunction(T value)
    {
        Console.WriteLine(value);
    }
}
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Integer Type");
        GenericClass<int> gcInt = new GenericClass<int>();
        gcInt.GenericFunction(200);

        Console.WriteLine("String Type");
        GenericClass<string> gcString =
               new GenericClass<string>();
        gcString.GenericFunction("This is a string");

        Console.ReadLine();
    }
}
```

T in the sample code above is not a reserved keyword. T, or any given name, means a type parameter.