Module 5

Chapter 19: Enumerating Collections

Consider an example for foreach statement to iterate the list of items in a simple array

```
int [ ] pins={10,20,30,40};
foreach(int x in pins)
{
        Console.WriteLine(x);
}
```

The foreach construct provides an elegant mechanism that greatly simplifies the code we need to write , by steping through enumerable collection, i.e it implements System.Collections.IEnumerable interface.

IEnumerable interface:

- It has a method **GetEnumerator ()**.
- The return type of the method is **System.Collections.IEnumerator**.
- IEnumerator has a property called **Current** and two methods **MoveNext ()** and **Reset ()**.

```
IEnumerable
{
        IEunumerator GetEnumerator ();
}
interface IEnumerator
{
        object Current
        {
                get;
        }
        bool MoveNext ();
        void Reset ();
}
```


Initially the pointer points before the first element.

MoveNext(): When we call MoveNext method to move the pointer down to next(first) item in the list and returns true if there is a another item if not it returns false.

Current: It is property used to access the item currently pointed to.

Reset (): It resets the pointer to point before the first element.

We observed that the current return object type which requires explicit type casting so the .Net framework provides generic enumerable collection “IEnumerable<T> and IEnumerator<T>”.

Manually implementing an enumerator

Tree.cs

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

namespace BinaryTree
{
   public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
   {
      public TItem NodeData { get; set; }
      public Tree<TItem> LeftTree { get; set; }
      public Tree<TItem> RightTree { get; set; }

      public Tree(TItem nodeValue)
      {
         this.NodeData = nodeValue;
         this.LeftTree = null;
         this.RightTree = null;
      }

      public void Insert(TItem newItem)
      {
         TItem currentNodeValue = this.NodeData;
         if (currentNodeValue.CompareTo(newItem) > 0)
         {
            // Insert the item into the left subtree
            if (this.LeftTree == null)
            {
               this.LeftTree = new Tree<TItem>(newItem);
            }
            else
            {
               this.LeftTree.Insert(newItem);
            }
         }
         else
         {
            // Insert the new item into the right subtree
            if (this.RightTree == null)
```

```
            {
                this.RightTree = new Tree<TItem>(newItem);
            }
            else
            {
                this.RightTree.Insert(newItem);
            }
        }
    }

    public string WalkTree()
    {
        string result = "";

        if (this.LeftTree != null)
        {
            result = this.LeftTree.WalkTree();
        }

        result += String.Format($" {this.NodeData.ToString()} ");

        if (this.RightTree != null)
        {
            result += this.RightTree.WalkTree();
        }

        return result;
    }

    IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
    {
        return new TreeEnumerator<TItem>(this);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
  }
}
```


Enumerator.cs

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

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

namespace BinaryTree
{
   class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
   {
      private Tree<TItem> currentData = null;
      private TItem currentItem = default(TItem);
      private Queue<TItem> enumData = null;

      public TreeEnumerator(Tree<TItem> data)
      {
         this.currentData = data;
      }

      object IEnumerator.Current
      {
         get
         {
            throw new NotImplementedException();
         }
      }

      TItem IEnumerator<TItem>.Current
      {
         get
         {
            if(this.enumData == null)
            {
               throw new InvalidOperationException("Use MoveNext before calling Current");
            }

            return this.currentItem;
         }
      }

      void IDisposable.Dispose()
      {
         // throw new NotImplementedException();
      }
```

```
    bool IEnumerator.MoveNext()
    {
      if(this.enumData == null)
      {
        this.enumData = new Queue<TItem>();
        populate(this.enumData, this.currentData);
      }

      if(this.enumData.Count > 0)
      {
        this.currentItem = this.enumData.Dequeue();
        return true;
      }

      return false;
    }

    private void populate(Queue<TItem> enumQueue, Tree<TItem> tree)
    {
      if (tree.LeftTree != null)
      {
        populate(enumQueue, tree.LeftTree);
      }

      enumQueue.Enqueue(tree.NodeData);

      if (tree.RightTree != null)
      {
        populate(enumQueue, tree.RightTree);
      }
    }

    void IEnumerator.Reset()
    {
      throw new NotImplementedException();
    }
  }
}
```

Program.cs

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

```
using System.Text;
using System.Threading.Tasks;
using BinaryTree;

namespace EnumeratorTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Tree<int> tree1 = new Tree<int>(10);
         tree1.Insert(5);
         tree1.Insert(11);
         tree1.Insert(5);
         tree1.Insert(-12);
         tree1.Insert(15);
         tree1.Insert(0);
         tree1.Insert(14);
         tree1.Insert(-8);
         tree1.Insert(10);

         foreach(int item in tree1)
         {
            Console.WriteLine(item);
         }
      }
   }
}
```


Implementing an enumerator by using an iterator:

- An iterator is a block of code that yields an ordered sequence of values.
- An iterator is not actually a member of an enumerable class. Rather it specifies the sequence that an enumerator should use for returning its values.
- The yield keyword indicates the value should be returned for each iteration.
- The yield statement halts the current iteration and passes the value to the caller.
- When the caller needs the next value, the GetEnumerator() continues at a point at which it left off, looping around and then yielding the next value.

  Ex:

```
class BasicCollection <T> : IEnumerable <T>
{
        private List <T> data = new List <T> ();
        public void FillList (params T [ ] item)
```

```
        {
                data.Add (item);
        }
        IEnumerator <T>  IEnumerable <T>.GetEnumerator ()
        {
                for (int i = 0; i < data.count; i++)
                {
                        yield return x;
                }
        }
    static void Main(string[] args)
                static void Main(string[] args)

        {
                BasicCollection <String> bc = new BuildCollection <String> ();
                bc.FillList ("Ram", "Sam", "Kumar");
                foreach (String x in bc)
                {
                        Console.WriteLine (x);
                }
        }
}
```

If we want to provide alternative iterarion mechanisms to present the data in a different sequence,we can implement additional properties that implement the Ienumerable interface and use that an iterator for returning data.(in this xample reverse property)


```
class BasicCollection <T> : IEnumerable <T>
    {
        private List <T> data = new List <T> ();
        public void FillList (params T [ ] item)
        {
                data.Add (item);
        }
        Public  IEnumerable<T> Reverse
        {
                Get
                {
                for (int i = data.Count-1; i >=0; i--)
                {
                        yield return data[i];
```

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

        {
            BasicCollection <String> bc = new BuildCollection <String> ();
            bc.FillList ("Ram", "Sam", "Kumar");
            foreach (String x in bc.Reverse)
            {
                Console.WriteLine (x);
            }
        }
}
```

Chapter 20
Operator Overloading

Operator Overloading:

Understanding operators:

- We use operators to combine into expressions. Each operators has its own semantics depending on the type with which it works.(ex + behalf like concatenate for string and other value types it will perform addition)
- Each operator has precedence. Ex: ***** has more precedence than **+ in a+b*c**.
- Each operator also has left associative and right associative ex a=b=c is evalauated as a=(b=c).
- Unary operator has one operand and binary has 2.
- Syntax:

```
public static ReturnType operator symbol (arguments)
{
        ……
        ……
}
```

- Constraints:
  - We cannot change the precedence and associativity of an operator it is based on the symbol not based on type.
    Ex : for **a+b*c**, first it performs b*c.
  - We cannot change the number of operands for a operator.
    Ex: **a++b** cannot be used.
  - We cannot invent any new operator. We can only overload for valid operator.
    Ex: ****** operator cannot be used.
  - We can't change meaning of the operator when they are applied on value types.
    Ex: **1+2**
  - We cannot overload **. (dot)** operator because **dot** operator is generally used for accessing class members.

Ex Program 1:

```
struct Hour
{
        private int value;
        public Hour (int v)
        {
                this.value = v;
        }
        public static Hour operator + (Hour l, Hour r)
```

```
        {
            return new Hour (l.value + r.value);
        }
        static void Main(string[] args)
        {
            Hour a = new Hour (5);
            Hour b = new Hour (2);
            Hour sum = a + b;
        }
    }
```

- Rules
  - It should be always public because all the operators are public.
  - It should be always static because it can never by polymorphic and cannot use the keywords like virtual, abstract, override or sealed modifiers.
  - Number of arguments should be matched. For binary operators, we need to pass 2 arguments, or for unary operators, we need to pass one argument and so on.
  - Since we are overloading for reference type at least one of the argument must always be of the containing type(reference type)

- **Symmetric**
  - We cannot add object and a value. So we need to use the constructor to convert the value type into reference type.

Ex 3:

```
struct Hour
{
    private int value;
    public Hour (int v)
    {
        this.value = v;
    }
    public static Hour operator + (Hour l, Hour r)
    {
        return new Hour (l.value + r.value);
    }

    public static Hour operator + (Hour l, int r)
```

```
        {
                return l+new Hour(r);
        }
        public static Hour operator + (int l, Hour r)
        {
                return new Hour(l)+r;
        }

        static void Main(string[] args)
        {
                Hour a = new Hour (5);
                int b = 2;
                Hour sum = a + b;
                Hour sum=b+a;
        }
}
```

Understanding compound statements:

- **Syntax:** a@=b;
  - where **'a'** is a object type
  - **'@'** is an operator
  - **'b'** can be object type or value type
- Example

  Hour a = new Hour (5);

  int b = 2;

  a += b; valid because a is object and b is value type

  b+=a; invalid no way to assign an Hour object to the built in type

Increment and decrement/comparing operators in structure and classes:

- If we use class, the reference of one object will be copied to another.

  Ex: Hour a = new Hour (5);

  Hour b = a++;

  Whatever updation is done to **'a'** will be affected to **'b'** also.
- In structure, since it is a value type, it will just assign the value and not the reference. So the updation done to **'a'** will not affect **'b'**.

  Ex:

```
struct Hour
{
        private int value;
        public Hour (int v)
```

```
        {
            this.value = v;
        }
        public static Hour operator ++ (Hour i)
        {
            value = i.value ++;
            return i;
        }
        static void Main(string[] args)
        {
            Hour a = new Hour (5);
            Hour postfix = a++;          //stmt1   (postfix = 5)
            Hour prefix = ++a;           //stmt2   (prefix = 7)
        }
    }
```

Explanation:

- In **stmt1**, we have performed postfix. So first the value of **'a**' is assigned to **'postfix'** and then **'a'** is incremented. Therefore after the execution of **stmt1**, the value of **'postfix'** is 5 and value of **'a'** is 6.
- In **stmt2**, we have performed prefix. So first the value of **'a'** is incremented and then it is assigned to prefix. Therefore, after the execution of **stmt2**, the value of **'prefix'** is 7 and the value of **'a'** is 7.
- It hour is a class then the prefix postfix and a all the values will be 7 since it is refrence to reference copy. To avoid this we need to change the definition of ++ operator as

```
public static Hour operator ++ (Hour i)
        {
            return new hour (i.value +1);

        }
```

in this implementation although updating done to one object will not affect the another but every time when we perform ++ operation a new object is created so this is expensive in terms of memory and garbage collection overhead.

Defining operator pairs:

C# enforces very reasonable expectation by insisting that if we overload equals (==) operator then we should overload not equals (!=) operator like < and > or <= and >= also.This is called as overloading with pairs.

The C# compiler doesn't write the code for any of these partners operator for u. so we should explicitly overload this pair operators.

Ex:

```
struct Hour
{
    private int value;
    public Hour (int v)
    {
        this.value = v;
    }
    public static bool operator == (Hour l, Hour r)
    {
        return (l.value == r.value);
    }
    public static bool operator != (Hour l, Hour r)
    {
        return (l.value != r.value);
    }
    static void Main(string[] args)
    {
        Hour a = new Hour (3);
        Hour b = new Hour (7);
        if (a == b)
            Console.WriteLine (“Equal”);
        else
            Console.WriteLine (“Not equal”);
    }
}
```


Understanding conversion operators

```
class Example
{
    public static void myDoublemethod(double x)
    {
        …….
        …….
    }
}
```

- In this example the method accepts the double value as argument. Suppose if we pass int value the compiler generates code that converts the value from int to double.

- This type of built in coversion is called implicit conversion or widening because the result argument is wider than original.

```
class Example
{
        public static void myIntmethod(int x)
        {
        }
        Example.myIntmethod(45.5);                    //error
}
```

- Here if we pass double value as a argument instead of int we will loss the information so the conversion is performed automatically. In this scenario we need to explicitly mention the type to which we need to convert. This is know as explicit conversion or narrowing because result is narrower than the original

```
Example.myIntmethod((int)45.5);
```

IMPLEMENTING USER-DEFINED CONVERSION OPERATORS

- The syntax of user defined conversion operator has some similarities to that of declaring an overloaded operator.
- But conversion must be public and it must be static, the type from which we are converting is declared as the parameter and to the type to which we are converting is declared as the type name after the keyword operator.

```
public static implicit operator int(Hour h)
{
        return h.value;
}
```


- Implicit conversion does required the casting.
- Explicit conversion requires the casting.
- But in operator over loading the conversion is always safe does not run the risk of losing information and cannot throw an exception so we can declare implicit or explicit.

Symmetric revisited

```
struct Hour
{
        private int value;
        public Hour (int v)
        {
```

```
                this.value = v;
        }
        public static Hour operator + (Hour l, Hour r)
        {
                return new Hour (l.value + r.value);
        }
        public static implicit operator hour(int h)
        {
                return new Hour( h);
        }
        static void Main(string[] args)
        {
                Hour a = new Hour (5);
                int b=3;
                Hour sum = a + b;
                Hour sum=b+a;
        }
}
```

Complex number program

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

namespace ComplexNumbers
{
        class Complex
        {
                public int Real { get; set; }
                public int Imaginary { get; set; }
                public Complex(int real, int imaginary)
                {
                        this.Real = real;
                        this.Imaginary = imaginary;
                }
                public Complex(int real)
                {
                        this.Real = real;
                        this.Imaginary = 0;
                }
```

```
public static implicit operator Complex(int from)
{
        return new Complex(from);
}
public static explicit operator int(Complex from)
{
        return from.Real;
}
public override string ToString()
{
        return $"({this.Real} + {this.Imaginary}i)";
}
public static Complex operator +(Complex lhs, Complex rhs)
{
        return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
}
public static Complex operator -(Complex lhs, Complex rhs)
{
        return new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
}
public static Complex operator *(Complex lhs, Complex rhs)
{
        return new Complex(lhs.Real * rhs.Real - lhs.Imaginary * rhs.Imaginary,
        lhs.Imaginary * rhs.Real + lhs.Real * rhs.Imaginary);
}
public static Complex operator /(Complex lhs, Complex rhs)
{
        int realElement = (lhs.Real * rhs.Real + lhs.Imaginary * rhs.Imaginary) /
                                (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);
        int imaginaryElement = (lhs.Imaginary * rhs.Real - lhs.Real *
                                        rhs.Imaginary) /(rhs.Real * rhs.Real +
                                        rhs.Imaginary * rhs.Imaginary);
        return new Complex(realElement, imaginaryElement);
}
public static bool operator ==(Complex lhs, Complex rhs)
{
        return lhs.Equals(rhs);
}
public static bool operator !=(Complex lhs, Complex rhs)
{
        return !(lhs.Equals(rhs));
}
public override bool Equals(object obj)
```

```
            {
                if(obj is Complex)
                {
                    Complex compare = (Complex)obj;
                    return (this.Real == compare.Real) && (this.Imaginary ==
                                        compare.Imaginary);
                }
                else
                {
                    return false;
                }
            }
            public override int GetHashCode()
            {
                return base.GetHashCode();
            }
        }
}
```

Program.cs

```
using System;

namespace ComplexNumbers
{
    class Program
    {
        static void doWork()
        {
            Complex first = new Complex(10, 4);
            Complex second = new Complex(5, 2);
            Console.WriteLine($"first is {first}");
            Console.WriteLine($"second is {second}");
            Complex temp = first + second;
            Console.WriteLine($"Add: result is {temp}");
            temp = first - second;
            Console.WriteLine($"Subtract: result is {temp}");
            temp = first * second;
            Console.WriteLine($"Multiply: result is {temp}");
            temp = first / second;
            Console.WriteLine($"Divide: result is {temp}");
            if(temp == first)
            {
                Console.WriteLine("Comparison: temp == first");
```

```
                }
                else
                {
                        Console.WriteLine("Comparison: temp != first");
                }
                if(temp == temp)
                {
                        Console.WriteLine("Comparison: temp == temp");
                }
                else
                {
                        Console.WriteLine("Comparison: temp != temp");
                }
                Console.WriteLine($"Current value of temp is {temp}");
                if(temp==2)
                {
                        Console.WriteLine("Comparison after conversion: temp == 2");
                }
                else
                {
                        Console.WriteLine("Comparison after conversion: temp != 2");
                }
                temp += 2;
                Console.WriteLine($"Value after adding 2: temp = {temp}");
                int tempInt = (int)temp;
                Console.WriteLine($"Int value after conversion: tempInt == {tempInt}");
        }
        static void Main()
        {
                try
                {
                        doWork();
                }
                catch (Exception ex)
                {
                        Console.WriteLine("Exception: {0}", ex.Message);
                }
                Console.ReadLine();
        }
    }
}
```

Chapter 21
LINQ (Language Integrated Query)

CUSTOMER

CID	FName	Lname	Company
1	Ram	Kumar	CITech
2	Sam	Peter	HP
3	Anil	Kumar	EMC^2
4	Anuj	Patil	Dell
5	Rahul	Jebin	HP
6	Kumar	Raj	CITech

ADDRESS

Company	House	Country
CITech	ABC	India
HP	XYZ	USA
EMC^2	MNO	UK
Dell	XBS	USA

Creating:

```
var CUSTOMER = new [ ]
{
        new {cid = "1", fname = "Ram", lname = "Kumar", company = "CITech"},
        new {cid = "2", fname = "Sam", lname = "Peter", company = "HP"},
        .
        .// like same for remaining rows

};
```


Retrieving:

- Using select method, we can retrieve specific data from the array by using lambda expression to define anonymous method
- The important things to understand:
  - The variable c is the parameter passed. It's an alias for each row in the customer array. The compiler deduces this from the fact that we are calling a select method on the customer array.
  - The select method does not actually retrieve the data at this time. It simply returns the enumerable object that will fetch the data identified by the select method when we iterate it.
  - The select method is not a method of the array type. It is an extension method of the enumerable class. It is stored in System.Linq.

Syntax:

object.Select (lambda expression);

Selecting a single column:

```
IEnumerable <String> Ans = CUSTOMER.Select (c => c.fname;);
```

Selecting multiple columns:

Three methods:

Method 1:

```
IEnumerable <String> Ans = CUSTOMER.Select ( c => $ "{c.fname} {c.lname}";};
foreach (String x in Ans)
{
    Console.WriteLine (x);
}
```

Method 2 – Selecting multiple columns using properties:

```
public class Names
{
    public Fname {get, set;}
    public Lname {get, set;}
}
IEnumerable <Names> name = CUSTOMER.Select (c => Fname = c.fname,
                                                 Lname = c.lname;);
```

Method 3 – Selecting multiple columns using anonymous object:

```
var Ans = CUSTOMER.Select (c => new { fn = c.fname, ln = c.lname} ;);
foreach (var x in Ans)
{
    Console.WriteLine ("fname = {0}, Lname = {1}", x.fn, x.ln);
}
```


Filtering data

With the select method, we can specify the fields that we want to include in enumerable selections or we can restrict the rows that enumerable collection should contain by using where method.

Where clause:

```
IEnumerable <String> Ans = Address.Where (addr => addr.String.Equals
                                    (addr.Country, "USA")).Select (addr =>
                                addr.company);
foreach (string x in Ans)
    {
```

```
        Console.WriteLine (x);
    }
```

Order by:

Ordering:

- To retrieve the data in a particular order, we can use the OrderBy method like the select and where methods, OrderBy expects a method as its arguments.
- This method identifies the expressions that we want to sort the data.
- If we want to sort the data on descending order, we need to use OrderBy Descending.

Ascending:

```
IEnumerable <String> Ans = CUSTOMER.OrderBy (c => c.fname).Select (c => c.fname);
```

Descending:

```
IEnumerable <String> Ans = CUSTOMER.OrderByDescending (c => c.fname).Select (c
                                                        =>c.fname);
```

Other way of writing queries (Easy Way)

Select:
```
var ans = from c in CUSTOMER Select new {c.fname, c.lname};
```

Order By:
```
var ans = from c in CUSTOMER
OrderBy c.fname
Select new {c.fname, c.lname};
```

Retrieve the CID, FName, LName and the country from the CUSTOMER and ADDRESS databases using join.

```
CUSTOMER.Select (c => new{c.cid, c.fname, c.lname})
.Join (Address, c => (c.Company, addr=> addr.Company,
(c, addr) => (c.cid, c.fname, c.lname, addr.country));
```

- Linq gives us the ability to join multiple sets of data over one or more common key fields.
- The parameters to the join methods are:
    - Enumerable Collections with which to join.
    - The method that identifies the common key fields from the data identified by the select method.
    - A method that identifies the common key fields on which to join the selected data.

- A method that specifies the columns that you require in the enumerable return sets written by join method.

Queries:

EMPLOYEE

EID	Ename	Designation	Company
001	Anuj	Manager	IBM
002	Raj	Emp	HP
003	Clinch	Emp	Dell
004	Kumar	GM	IBM
005	Sam	Emp	HP
006	Ganesh	Emp	HP

LINQ Queries:

```
var ans = Employee,GroupBy (E => e.Company);
foreach (var x in ans)
{
        Console.WriteLine ("Company Name {0}", x.key);
        Console.WriteLine ("Company Name {0}", x.count);        //key and count are
built in variables, key holds the grouping attribute and count holds the number of items
in each group
        foreach (var i in x)
                Console.WriteLine (i.ename);
}
```

QUERY OPERATION

- Since the LINQ syntax make use of several advanced C# language features, and resultant code can sometimes be quite hard to understand and maintain.
- To relieve this burden the designer of c# added query operation to the language with which we can employ LINQ features by using a syntax more akin to SQL.

Simplified Query operations using from:

1. var ans=from c in Customer select c.fname;
2. var ans=from c in customer
   where String.equals(c.country,"USA")
   select c.fname;
3. var ans=from c in Customer
   order by c.fname
   select c.fname;
4. var ans=from e in Employee
   group e by e.company;
5. int no_of_company = (from e in Employee
   select e.company).distinct().count();

// to select everything like *

6. int ans=from e in Employee select e;
7. var ans=from a in address
   join c in customer
   on a.company equals c.company
   select new{c.fname,c.lame,a.country};

LINQ and deferred evaluation:

1. var ans = from e in Employee
   Select e.name, employee [3].ename = “Ranesh”;

- When we use LINQ on query operation to define an enumerable collection, if we execute any query, only the objects are returned but it is not enumerable.
- The collection is enumerated only when we iterate it.
- If we change ant data, the recent updated one is used to iterate.
- Whenever we use ToList () method, the old table entry is stored in the cache memory. So when we iterate, we fetch the old data.
- var ans = from e in Employee ToList ()
  Select e.ename, employee[3].ename = “Ramesh”;