

Lab Manual

Technology Management

A simplified practical work book of System Programming course for Computer Science, Information Technology and Software Engineering students

Student Name: _____

Registration No: _____

Section / Semester: _____

Submission Date: _____

Student Signature: _____

Marks Obtained: _____

Maximum Marks: _____

Examiner Name/Sig: _____

Mr. Shahid Jamil



Lab Manual System Programming: First Edition

Author(s): Mr. Shahid Jamil

Publication Date: 23/3/2018

A publication of BIIT Series

Copyrights © Reserved By Authors

Learning Objectives: By the end of the lab work, students should have following skills;

Sr.	Learning Objective	LAB No.
1	Introduction to Collections & Generics	1
2	Array list & Hash table	2
3	Sorted list & Stack	3
4	Queue & Bit Array	4
5	Generic list & Generic Dictionary	5
6	Overriding Functions	6
7	Introduction To LINQ	7
8	LINQ (Method Syntax)	8
9	Delegates	9
10	Multithreading	10
11	Thread Synchronization	11
12	Socket Programming	12
13	Design Patterns	13

Table of Contents

Lab No 1:.....	8
Lab No 2:.....	9
Lab No 3:.....	27
Lab No 4:.....	35
Lab No 5:.....	44
Lab No 6:.....	45
Lab No 7:.....	46
LINQ (Language-Integrated Query).....	46
Advantages of LINQ	46
Need For LINQ.....	47
Types of LINQ	48
LINQ Architecture in .NET	48
Difference between LINQ and Stored Procedure	48
LINQ - Query Operators	49
Filtering Operators	50
Join Operators.....	50
Projection Operations.....	51
Sorting Operators	51
Grouping Operators.....	52
Conversions.....	52
Concatenation	53
Aggregation.....	54
Quantifier Operations.....	54
Partition Operators.....	55
Generation Operations.....	56
Set Operations.....	56
Equality.....	57
Element Operators.....	57
Filtering Operators - Where	58
Where	59
Where clause in Query Syntax:	59
Where extension method in Method Syntax:.....	61
Multiple Where clause:	62
Sorting Operators: OrderBy & OrderByDescending	62
OrderBy:.....	63
OrderBy in Method Syntax:	64
OrderByDescending:	65
Multiple Sorting:	66

Points to Remember:	67
Sorting Operators: ThenBy & ThenByDescending	67
Points to Remember :	69
Grouping Operators: GroupBy & ToLookup	69
GroupBy:	69
GroupBy in Query Syntax:	70
GroupBy in Method Syntax:	72
ToLookup	73
Points to Remember :	74
Joining Operator: Join	74
Join:	74
Join in Method Syntax:	74
Join in Query Syntax:	77
Projection Operators: Select, SelectMany	79
Select:	79
Select in Query Syntax:	79
Select in Method Syntax:	81
Aggregation Operators: Aggregate	81
Aggregate:	82
Aggregate method with seed value:	83
Aggregate method with result selector:	84
Element Operators: First & FirstOrDefault	85
Element Operators : Last & LastOrDefault	87
Partitioning Operators: Skip & SkipWhile	89
Skip:	90
Skip operator in Query Syntax:	90
SkipWhile:	90
Skip/SkipWhile operator in Query Syntax:	93
Partitioning Operators: Take & TakeWhile	93
Take:	94
TakeWhile:	94
Delegates	96
Declaring Delegates	96
Action and Func Delegates in C#	96
Multithreading in C#	98
Threads and Thread Synchronization in C#	101
Introduction	101
Multithreading Fundamentals	101
Creating a Thread	102
Example	102

Methods.....	103
States of a Thread	104
Properties of a Thread	104
PriorityLevels of Thread	105
Synchronization in Threads	105
Using the Lock Keyword	106
Converting the Code to Enable Synchronization using the Lock Keyword	106
Using the Monitor Type.....	107
TcpClient Class	107
Inheritance Hierarchy	108
Syntax	108
Constructors	108
Properties	108
Methods	109
Remarks	110
Notes to Inheritors:	110
Examples	111
Security	111
TcpListener Class	112
Syntax	112
Constructors	112
Properties	112
Methods	113
Remarks	114
Examples	114
UdpClient Class	116
Syntax	116
Constructors	116
Properties	116
Methods	117
Remarks	119
Examples	119
Security	120
Singleton	120
Definition.....	120

UML class diagram.....	121
Participants	121
Structural code in C#	121
Real-world code in C#	123
.NET Optimized code in C#.....	126
Abstract Factory	129
Definition.....	129
UML class diagram.....	129
Participants	130
Structural code in C#	130
Real-world code in C#	134
Adapter	137
Definition.....	137
UML class diagram.....	137
Participants	138
Structural code in C#	138
Real-world code in C#	140
Composite.....	144
Definition.....	144
UML class diagram.....	144
Participants	145
Structural code in C#	146
Real-world code in C#	149
Command	152
Definition.....	152
UML class diagram.....	152
Participants	153
Structural code in C#	153
Real-world code in C#	156
Interpreter	160
Definition.....	160
UML class diagram.....	160
Participants	161
Structural code in C#	162
Real-world code in C#	164

Lab No 1:

Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

Various Collection Classes and Their Usage

The following are the various commonly used classes of the System.Collection namespace. Click the following links to check their detail.

Sr.No.	Class & Description and Usages
1	<u>ArrayList</u> It represents ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.
2	<u>Hashtable</u> It uses a key to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.
3	<u>SortedList</u> It uses a key as well as an index to access the items in a list. A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.

4	<u>Stack</u> It represents a last-in, first out collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.
5	<u>Queue</u> It represents a first-in, first out collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue and when you remove an item, it is called deque.
6	<u>BitArray</u> It represents an array of the binary representation using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.



Lab No 2:

















ARRAYLIST:





Methods

	Name	Description
	<u>Adapter(ICollection)</u>	Creates an <u>ArrayList</u> wrapper for a specific <u>ICollection</u> .
	<u>Add(Object)</u>	Adds an object to the end of the <u>ArrayList</u> .
	<u>AddRange(ICollection)</u>	Adds the elements of an <u>ICollection</u> to the end of the <u>ArrayList</u> .
	<u>BinarySearch(Int32, Int32, Object, IComparer)</u>	Searches a range of elements in the sorted <u>ArrayList</u> for an element using the specified comparer and returns the zero-based index of the element.








<u>BinarySearch(Object)</u>	Searches the entire sorted <u>ArrayList</u> for an element using the default comparer and returns the zero-based index of the element.
<u>BinarySearch(Object, IComparer)</u>	Searches the entire sorted <u>ArrayList</u> for an element using the specified comparer and returns the zero-based index of the element.
<u>Clear()</u>	Removes all elements from the <u>ArrayList</u> .
<u>Clone()</u>	Creates a shallow copy of the <u>ArrayList</u> .
<u>Contains(Object)</u>	Determines whether an element is in the <u>ArrayList</u> .
<u>CopyTo(Array)</u>	Copies the entire <u>ArrayList</u> to a compatible one-dimensional <u>Array</u> , starting at the beginning of the target array.
<u>CopyTo(Array, Int32)</u>	Copies the entire <u>ArrayList</u> to a compatible one-dimensional <u>Array</u> , starting at the specified index of the target array.
<u>CopyTo(Int32, Array, Int32, Int32)</u>	Copies a range of elements from the <u>ArrayList</u> to a compatible one-dimensional <u>Array</u> , starting at the specified index of the target array.
<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)
<u>FixedSize(ArrayList)</u>	Returns an <u>ArrayList</u> wrapper with a fixed size.
<u>FixedSize(ICollection)</u>	Returns an <u>ICollection</u> wrapper with a fixed size.

	<u>GetEnumerator()</u>	Returns an enumerator for the entire <u>ArrayList</u> .
	<u>GetEnumerator(Int32, Int32)</u>	Returns an enumerator for a range of elements in the <u>ArrayList</u> .
	<u>GetHashCode()</u>	Serves as the default hash function. (Inherited from <u>Object</u> .)
	<u>GetRange(Int32, Int32)</u>	Returns an <u>ArrayList</u> which represents a subset of the elements in the source <u>ArrayList</u> .
	<u>GetType()</u>	Gets the <u>Type</u> of the current instance.(Inherited from <u>Object</u> .)
	<u>IndexOf(Object)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the first occurrence within the entire <u>ArrayList</u> .
	<u>IndexOf(Object, Int32)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the first occurrence within the range of elements in the <u>ArrayList</u> that extends from the specified index to the last element.
	<u>IndexOf(Object, Int32, Int32)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the first occurrence within the range of elements in the <u>ArrayList</u> that starts at the specified index and contains the specified number of elements.
	<u>Insert(Int32, Object)</u>	Inserts an element into the <u>ArrayList</u> at the specified index.
	<u>InsertRange(Int32, ICollection)</u>	Inserts the elements of a collection into the <u>ArrayList</u> at the specified index.
	<u>LastIndexOf(Object)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the last occurrence within the entire <u>ArrayList</u> .
	<u>LastIndexOf(Object, Int32)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the last occurrence within the range of elements in the <u>ArrayList</u> that extends from the first element to the specified index.

	<u>LastIndexOf(Object, Int32, Int32)</u>	Searches for the specified <u>Object</u> and returns the zero-based index of the last occurrence within the range of elements in the <u>ArrayList</u> that contains the specified number of elements and ends at the specified index.
	<u>MemberwiseClone()</u>	Creates a shallow copy of the current <u>Object</u> . (Inherited from <u>Object</u> .)
	<u>ReadOnly(ArrayList)</u>	Returns a read-only <u>ArrayList</u> wrapper.
	<u>ReadOnly(ICollection)</u>	Returns a read-only <u>ICollection</u> wrapper.
	<u>Remove(Object)</u>	Removes the first occurrence of a specific object from the <u>ArrayList</u> .
	<u>RemoveAt(Int32)</u>	Removes the element at the specified index of the <u>ArrayList</u> .
	<u>RemoveRange(Int32, Int32)</u>	Removes a range of elements from the <u>ArrayList</u> .
	<u>Repeat(Object, Int32)</u>	Returns an <u>ArrayList</u> whose elements are copies of the specified value.
	<u>Reverse()</u>	Reverses the order of the elements in the entire <u>ArrayList</u> .
	<u>Reverse(Int32, Int32)</u>	Reverses the order of the elements in the specified range.
	<u>SetRange(Int32, ICollection)</u>	Copies the elements of a collection over a range of elements in the <u>ArrayList</u> .
	<u>Sort()</u>	Sorts the elements in the entire <u>ArrayList</u> .
	<u>Sort(IComparer)</u>	Sorts the elements in the entire <u>ArrayList</u> using the specified comparer.
	<u>Sort(Int32, Int32, IComparer)</u>	Sorts the elements in a range of elements in <u>ArrayList</u> using the specified comparer.
	<u>Synchronized(ArrayList)</u>	Returns an <u>ArrayList</u> wrapper that is synchronized (thread safe).
	<u>Synchronized(ICollection)</u>	Returns an <u>ICollection</u> wrapper that is synchronized (thread

		safe).
	ToArray()	Copies the elements of the ArrayList to a new Object array.
	ToArray(Type)	Copies the elements of the ArrayList to a new array of the specified element type.
	ToString()	Returns a string that represents the current object.(Inherited from Object .)
	TrimToSize()	Sets the capacity to the actual number of elements in the ArrayList .

Properties

	Name	Description
	Capacity	Gets or sets the number of elements that the ArrayList can contain.
	Count	Gets the number of elements actually contained in the ArrayList .
	IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
	IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
	IsSynchronized	Gets a value indicating whether access to the ArrayList is synchronized (thread safe).
	Item[Int32]	Gets or sets the element at the specified index.
	SyncRoot	Gets an object that can be used to synchronize access to the ArrayList .

Examples

C# program that uses ArrayList

```
using System.Collections;
```

```
class Program
{
```

```

static void Main()
{
    // Create an ArrayList and add 3 elements.
    ArrayList list = new ArrayList();
    list.Add("One");
    list.Add("Two");
    list.Add("Three");
}
}

```

C# program that uses ArrayList parameter

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList and add two ints.
        //
        ArrayList list = new ArrayList();
        list.Add(5);
        list.Add(7);
        //
        // Use ArrayList with method.
        //
        Example(list);
    }

    static void Example(ArrayList list)
    {
        foreach (int i in list)
        {
            Console.WriteLine(i);
        }
    }
}

```

Output

```

5
7

```

C# program that uses Add and AddRange

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList with two values.
        //
        ArrayList list = new ArrayList();
        list.Add(5);
        list.Add(7);
        //
        // Second ArrayList.
        //
        ArrayList list2 = new ArrayList();
        list2.Add(10);
        list2.Add(13);
        //
        // Add second ArrayList to first.
        //
        list.AddRange(list2);
        //
        // Display the values.
        //
        foreach (int i in list)
        {
            Console.WriteLine(i);
        }
    }
}

```

Output

```

5
7
10
13

```

C# program that uses Count

```

using System;
using System.Collections;

class Program
{
    static void Main()

```

```

{
    //
    // Create an ArrayList with two values.
    //
    ArrayList list = new ArrayList();
    list.Add(9);
    list.Add(10);
    //
    // Show number of elements in ArrayList.
    //
    Console.WriteLine(list.Count);
    //
    // Clear the ArrayList.
    //
    list.Clear();
    //
    // Show count again.
    //
    Console.WriteLine(list.Count);
}
}

```

Output

2
0

C# program that sorts ArrayList and reverses

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList with four strings.
        //
        ArrayList list = new ArrayList();
        list.Add("Cat");
        list.Add("Zebra");
        list.Add("Dog");
        list.Add("Cow");
        //
        // Sort the ArrayList.
        //
        list.Sort();
    }
}

```



```

//
// Display the ArrayList elements.
//
foreach (string value in list)
{
    Console.WriteLine(value);
}
//
// Reverse the ArrayList.
//
list.Reverse();
//
// Display the ArrayList elements again.
//
foreach (string value in list)
{
    Console.WriteLine(value);
}
}
}

```

Output

Cat
 Cow
 Dog
 Zebra

Zebra
 Dog
 Cow
 Cat

C# program that uses Insert and Remove

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList with three strings.
        //
        ArrayList list = new ArrayList();
        list.Add("Dot");
        list.Add("Net");
    }
}

```

```

list.Add("Perls");
//
// Remove middle element in ArrayList.
//
list.RemoveAt(1); // It becomes [Dot, Perls]
//
// Insert word at the beginning of ArrayList.
//
list.Insert(0, "Carrot"); // It becomes [Carrot, Dot, Perls]
//
// Remove first two words from ArrayList.
//
list.RemoveRange(0, 2);
//
// Display the result ArrayList.
//
foreach (string value in list)
{
    Console.WriteLine(value); // <-- "Perls"
}
}
}

```

Output

Perls

C# that uses ArrayList and for

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList with three strings.
        //
        ArrayList list = new ArrayList();
        list.Add("man");
        list.Add("woman");
        list.Add("plant");
        //
        // Loop over ArrayList.
        //
        for (int i = 0; i < list.Count; i++)
        {

```

```

        string value = list[i] as string;
        Console.WriteLine(value);
    }
}

```

Output

man
woman
plant

C# that uses GetRange

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Create an ArrayList with 4 strings.
        //
        ArrayList list = new ArrayList();
        list.Add("fish");
        list.Add("amphibian");
        list.Add("bird");
        list.Add("plant");
        //
        // Get last two elements in ArrayList.
        //
        ArrayList range = list.GetRange(2, 2);
        //
        // Display the elements.
        //
        foreach (string value in range)
        {
            Console.WriteLine(value); // bird, plant
        }
    }
}














```








Output

bird
plant







HASHTABLE:






Methods

	Name	Description
	<u>Add(Object, Object)</u>	Adds an element with the specified key and value into the <u>Hashtable</u> .
	<u>Clear()</u>	Removes all elements from the <u>Hashtable</u> .
	<u>Clone()</u>	Creates a shallow copy of the <u>Hashtable</u> .
	<u>Contains(Object)</u>	Determines whether the <u>Hashtable</u> contains a specific key.
	<u>ContainsKey(Object)</u>	Determines whether the <u>Hashtable</u> contains a specific key.
	<u>ContainsValue(Object)</u>	Determines whether the <u>Hashtable</u> contains a specific value.
	<u>CopyTo(Array, Int32)</u>	Copies the <u>Hashtable</u> elements to a one-dimensional <u>Array</u> instance at the specified index.
	<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
	<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)
	<u>GetEnumerator()</u>	Returns an <u>IDictionaryEnumerator</u> that iterates through the <u>Hashtable</u> .
	<u>GetHash(Object)</u>	Returns the hash code for the specified key.
	<u>GetHashCode()</u>	Serves as the default hash function. (Inherited from <u>Object</u> .)
	<u>GetObjectData(SerializationInfo, StreamingContext)</u>	Implements the <u>ISerializable</u> interface and returns the data needed to serialize the <u>Hashtable</u> .

	<u>GetType()</u>	Gets the <u>Type</u> of the current instance.(Inherited from <u>Object</u> .)
	<u>KeyEquals(Object, Object)</u>	Compares a specific <u>Object</u> with a specific key in the <u>Hashtable</u> .
	<u>MemberwiseClone()</u>	Creates a shallow copy of the current <u>Object</u> .(Inherited from <u>Object</u> .)
	<u>OnDeserialization(Object)</u>	Implements the <u>ISerializable</u> interface and raises the deserialization event when the deserialization is complete.
	<u>Remove(Object)</u>	Removes the element with the specified key from the <u>Hashtable</u> .
	<u>Synchronized(Hashtable)</u>	Returns a synchronized (thread-safe) wrapper for the <u>Hashtable</u> .
	<u>ToString()</u>	Returns a string that represents the current object.(Inherited from <u>Object</u> .)

Properties

	Name	Description
	<u>comparer</u>	Obsolete. Gets or sets the <u>IComparer</u> to use for the <u>Hashtable</u> .
	<u>Count</u>	Gets the number of key/value pairs contained in the <u>Hashtable</u> .
	<u>EqualityComparer</u>	Gets the <u>IEqualityComparer</u> to use for the <u>Hashtable</u> .
	<u>hcp</u>	Obsolete. Gets or sets the object that can dispense hash codes.
	<u>IsFixedSize</u>	Gets a value indicating whether the <u>Hashtable</u> has a fixed size.
	<u>IsReadOnly</u>	Gets a value indicating whether the <u>Hashtable</u> is read-only.

	IsSynchronized	Gets a value indicating whether access to the Hashtable is synchronized (thread safe).
	Item[Object]	Gets or sets the value associated with the specified key.
	Keys	Gets an ICollection containing the keys in the Hashtable .
	SyncRoot	Gets an object that can be used to synchronize access to the Hashtable .
	Values	Gets an ICollection containing the values in the Hashtable .

Examples

C# program that adds entries to Hashtable

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        Hashtable hashtable = new Hashtable();
        hashtable[1] = "One";
        hashtable[2] = "Two";
        hashtable[13] = "Thirteen";

        foreach (DictionaryEntry entry in hashtable)
        {
            Console.WriteLine("{0}, {1}", entry.Key, entry.Value);
        }
    }
}
```

Output

```
13, Thirteen
2, Two
1, One
```

C# program that uses Contains method

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

```
class Program
```

```

{
    static Hashtable GetHashtable()
    {
        // Create and return new Hashtable.
        Hashtable hashtable = new Hashtable();
        hashtable.Add("Area", 1000);
        hashtable.Add("Perimeter", 55);
        hashtable.Add("Mortgage", 540);
        return hashtable;
    }

    static void Main()
    {
        Hashtable hashtable = GetHashtable();

        // See if the Hashtable contains this key.
        Console.WriteLine(hashtable.ContainsKey("Perimeter"));

        // Test the Contains method.
        // ... It works the same way.
        Console.WriteLine(hashtable.Contains("Area"));

        // Get value of Area with indexer.
        int value = (int)hashtable["Area"];

        // Write the value of Area.
        Console.WriteLine(value);
    }
}

```

Output

```

True
True
1000

```

C# program that uses multiple types

```

using System;
using System.Collections;

class Program
{
    static Hashtable GetHashtable()
    {
        Hashtable hashtable = new Hashtable();

        hashtable.Add(300, "Carrot");
    }
}

```

```

        hashtable.Add("Area", 1000);
        return hashtable;
    }

    static void Main()
    {
        Hashtable hashtable = GetHashtable();

        string value1 = (string)hashtable[300];
        Console.WriteLine(value1);

        int value2 = (int)hashtable["Area"];
        Console.WriteLine(value2);
    }
}

```

Output

Carrot
1000

C# program that casts Hashtable values

```

using System;
using System.Collections;
using System.IO;

class Program
{
    static void Main()
    {
        Hashtable hashtable = new Hashtable();
        hashtable.Add(400, "Blazer");

        // This cast will succeed.
        string value = hashtable[400] as string;
        if (value != null)
        {
            Console.WriteLine(value);
        }

        // This cast won't succeed, but won't throw.
        StreamReader reader = hashtable[400] as StreamReader;
        if (reader != null)
        {
            Console.WriteLine("Unexpected");
        }
    }
}

```



```

// You can get the object and test it.
object value2 = hashtable[400];
if (value2 is string)
{
    Console.Write("is string: ");
    Console.WriteLine(value2);
}
}
}

```

Output

Blazer
is string: Blazer

C# program that loops over Keys, Values

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        Hashtable hashtable = new Hashtable();
        hashtable.Add(400, "Blaze");
        hashtable.Add(500, "Fiery");
        hashtable.Add(600, "Fire");
        hashtable.Add(800, "Immolate");

        // Display the keys.
        foreach (int key in hashtable.Keys)
        {
            Console.WriteLine(key);
        }

        // Display the values.
        foreach (string value in hashtable.Values)
        {
            Console.WriteLine(value);
        }

        // Put keys in an ArrayList.
        ArrayList arrayList = new ArrayList(hashtable.Keys);
        foreach (int key in arrayList)
        {
            Console.WriteLine(key);
        }
    }
}

```

```
    }  
  }  
}
```

Output

```
800    (First loop)  
600  
500  
400  
Immolate (Second loop)  
Fire  
Fiery  
Blaze  
800    (Third loop)  
600  
500  
400
```

C# program that uses Count

```
using System;  
using System.Collections;  
  
class Program  
{  
    static void Main()  
    {  
        // Add four elements to Hashtable.  
        Hashtable hashtable = new Hashtable();  
        hashtable.Add(1, "Sandy");  
        hashtable.Add(2, "Bruce");  
        hashtable.Add(3, "Fourth");  
        hashtable.Add(10, "July");  
  
        // Get Count of Hashtable.  
        int count = hashtable.Count;  
        Console.WriteLine(count);  
  
        // Clear the Hashtable.  
        hashtable.Clear();  
  
        // Get Count of Hashtable again.  
        Console.WriteLine(hashtable.Count);  
    }  
}
```














Output













4
0

Lab No 3:




SORTEDLIST:







Methods

	Name	Description
	<u>Add(Object, Object)</u>	Adds an element with the specified key and value to a <u>SortedList</u> object.
	<u>Clear()</u>	Removes all elements from a <u>SortedList</u> object.
	<u>Clone()</u>	Creates a shallow copy of a <u>SortedList</u> object.
	<u>Contains(Object)</u>	Determines whether a <u>SortedList</u> object contains a specific key.
	<u>ContainsKey(Object)</u>	Determines whether a <u>SortedList</u> object contains a specific key.
	<u>ContainsValue(Object)</u>	Determines whether a <u>SortedList</u> object contains a specific value.
	<u>CopyTo(Array, Int32)</u>	Copies <u>SortedList</u> elements to a one-dimensional <u>Array</u> object, starting at the specified index in the array.
	<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
	<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)
	<u>GetByIndex(Int32)</u>	Gets the value at the specified index of a <u>SortedList</u> object.
	<u>GetEnumerator()</u>	Returns an <u>IDictionaryEnumerator</u> object that iterates through a <u>SortedList</u> object.
	<u>GetHashCode()</u>	Serves as the default hash function. (Inherited from <u>Object</u> .)
	<u>GetKey(Int32)</u>	Gets the key at the specified index of a <u>SortedList</u> object.

	<u>GetKeyList()</u>	Gets the keys in a <u>SortedList</u> object.
	<u>GetType()</u>	Gets the <u>Type</u> of the current instance.(Inherited from <u>Object</u> .)
	<u>GetValueList()</u>	Gets the values in a <u>SortedList</u> object.
	<u>IndexOfKey(Object)</u>	Returns the zero-based index of the specified key in a <u>SortedList</u> object.
	<u>IndexOfValue(Object)</u>	Returns the zero-based index of the first occurrence of the specified value in a <u>SortedList</u> object.
	<u>MemberwiseClone()</u>	Creates a shallow copy of the current <u>Object</u> .(Inherited from <u>Object</u> .)
	<u>Remove(Object)</u>	Removes the element with the specified key from a <u>SortedList</u> object.
	<u>RemoveAt(Int32)</u>	Removes the element at the specified index of a <u>SortedList</u> object.
	<u>SetByIndex(Int32, Object)</u>	Replaces the value at a specific index in a <u>SortedList</u> object.
	<u>Synchronized(SortedList)</u>	Returns a synchronized (thread-safe) wrapper for a <u>SortedList</u> object.
	<u>ToString()</u>	Returns a string that represents the current object.(Inherited from <u>Object</u> .)
	<u>TrimToSize()</u>	Sets the capacity to the actual number of elements in a <u>SortedList</u> object.

Properties

	Name	Description
	<u>Capacity</u>	Gets or sets the capacity of a <u>SortedList</u> object.
	<u>Count</u>	Gets the number of elements contained in a <u>SortedList</u> object.
	<u>IsFixedSize</u>	Gets a value indicating whether a <u>SortedList</u> object has a fixed size.

	IsReadOnly	Gets a value indicating whether a SortedList object is read-only.
	IsSynchronized	Gets a value indicating whether access to a SortedList object is synchronized (thread safe).
	Item[Object]	Gets and sets the value associated with a specific key in a SortedList object.
	Keys	Gets the keys in a SortedList object.
	SyncRoot	Gets an object that can be used to synchronize access to a SortedList object.
	Values	Gets the values in a SortedList object.

Examples

C# program that uses SortedList

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

class Program
{
    static void Main()
    {
        //
        // Created SortedList with three keys and values.
        //
        SortedList<string, int> sorted = new SortedList<string, int>();
        sorted.Add("perls", 3);
        sorted.Add("dot", 1);
        sorted.Add("net", 2);

        //
        // Test SortedList with ContainsKey method.
        //
        bool contains1 = sorted.ContainsKey("java");
        Console.WriteLine("contains java = " + contains1);

        //
        // Use TryGetValue method.
        //
        int value;
        if (sorted.TryGetValue("perls", out value))
        {
            Console.WriteLine("perls key is = " + value);
        }
    }
}
```

```

    }

    //
    // Use item indexer.
    //
    Console.WriteLine("dot key is = " + sorted["dot"]);

    //
    // Loop over SortedList data.
    //
    foreach (var pair in sorted)
    {
        Console.WriteLine(pair);
    }

    //
    // Get index of key and then index of value.
    //
    int index1 = sorted.IndexOfKey("net");
    Console.WriteLine("index of net (key) = " + index1);

    int index2 = sorted.IndexOfValue(3);
    Console.WriteLine("index of 3 (value) = " + index2);

    //
    // Display Count property.
    //
    Console.WriteLine("count is = " + sorted.Count);
}
}

```

Output

















```

contains java = False
perls key is = 3
dot key is = 1
[dot, 1]
[net, 2]
[perls, 3]
index of net (key) = 1
index of 3 (value) = 2
count is = 3




```

STACK:

Methods

	Name	Description
	<u>Clear()</u>	Removes all objects from the <u>Stack</u> .
	<u>Clone()</u>	Creates a shallow copy of the <u>Stack</u> .
	<u>Contains(Object)</u>	Determines whether an element is in the <u>Stack</u> .
	<u>CopyTo(Array, Int32)</u>	Copies the <u>Stack</u> to an existing one-dimensional <u>Array</u> , starting at the specified array index.
	<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
	<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)
	<u>GetEnumerator()</u>	Returns an <u>IEnumerator</u> for the <u>Stack</u> .
	<u>GetHashCode()</u>	Serves as the default hash function. (Inherited from <u>Object</u> .)
	<u>GetType()</u>	Gets the <u>Type</u> of the current instance.(Inherited from <u>Object</u> .)
	<u>MemberwiseClone()</u>	Creates a shallow copy of the current <u>Object</u> .(Inherited from <u>Object</u> .)
	<u>Peek()</u>	Returns the object at the top of the <u>Stack</u> without removing it.
	<u>Pop()</u>	Removes and returns the object at the top of the <u>Stack</u> .
	<u>Push(Object)</u>	Inserts an object at the top of the <u>Stack</u> .
	<u>Synchronized(Stack)</u>	Returns a synchronized (thread safe) wrapper for the <u>Stack</u> .
	<u>ToArray()</u>	Copies the <u>Stack</u> to a new array.
	<u>ToString()</u>	Returns a string that represents the current object.(Inherited from <u>Object</u> .)

Properties

	Name	Description
	Count	Gets the number of elements contained in the Stack .
	IsSynchronized	Gets a value indicating whether access to the Stack is synchronized (thread safe).
	SyncRoot	Gets an object that can be used to synchronize access to the Stack .

Examples

C# program that creates new Stack of integers

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

class Program
{
    static Stack<int> GetStack()
    {
        Stack<int> stack = new Stack<int>();
        stack.Push(100);
        stack.Push(1000);
        stack.Push(10000);
        return stack;
    }

    static void Main()
    {
        var stack = GetStack();
        Console.WriteLine("--- Stack contents ---");
        foreach (int i in stack)
        {
            Console.WriteLine(i);
        }
    }
}
```

Output

```
--- Stack contents ---
10000
1000
100
```


C# program that uses Pop method

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

class Program
{
    static void Main()
    {
        // Get the stack.
        // ... See above definition of this method.
        Stack<int> stack = GetStack();

        // Pop the top element.
        int pop = stack.Pop();

        // Write to the console.
        Console.WriteLine("--- Element popped from top of Stack ---");
        Console.WriteLine(pop);

        // Now look at the top element.
        int peek = stack.Peek();
        Console.WriteLine("--- Element now at the top ---");
        Console.WriteLine(peek);
    }
}
```

Output

```
--- Element popped from top of Stack ---
10000
--- Element now at the top ---
1000
```

C# program that uses Clear and Count methods

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

class Program
{
    static void Main()
    {
        // Get the stack [See method definition above]
        Stack<int> stack = GetStack();
```

```

// Count the number of elements in the Stack
int count = stack.Count;
Console.WriteLine("--- Element count ---");
Console.WriteLine(count);

// Clear the Stack
stack.Clear();
Console.WriteLine("--- Stack was cleared ---");
Console.WriteLine(stack.Count);
}
}

```

Output

```

--- Element count ---
3
--- Stack was cleared ---
0

```

C# program that uses Stack incorrectly, correctly

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Create an empty Stack.
        var stack = new Stack<int>();

        try
        {
            // This throws an exception.
            int pop = stack.Pop();
        }
        catch (Exception ex)
        {
            Console.WriteLine("--- Exception raised by Pop ---");
            Console.WriteLine(ex.ToString());
        }

        // Here we safely Pop the stack.
        if (stack.Count > 0)
        {
            int safe = stack.Pop();
        }
    }
}

```

```

else
{
    Console.WriteLine("--- Avoided exception by using Count method ---");
}
}
}

```

Output

```









--- Exception raised by Pop ---
System.InvalidOperationException: Stack empty.
...
...
--- Avoided exception by using Count method










```

Lab No 4:




QUEUE :

Methods

	Name	Description
	<u>Clear()</u>	Removes all objects from the <u>Queue</u> .
	<u>Clone()</u>	Creates a shallow copy of the <u>Queue</u> .
	<u>Contains(Object)</u>	Determines whether an element is in the <u>Queue</u> .
	<u>CopyTo(Array, Int32)</u>	Copies the <u>Queue</u> elements to an existing one-dimensional <u>Array</u> , starting at the specified array index.
	<u>Dequeue()</u>	Removes and returns the object at the beginning of the <u>Queue</u> .
	<u>Enqueue(Object)</u>	Adds an object to the end of the <u>Queue</u> .
	<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
	<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)

	GetEnumerator()	Returns an enumerator that iterates through the Queue .
	GetHashCode()	Serves as the default hash function. (Inherited from Object .)
	GetType()	Gets the Type of the current instance.(Inherited from Object .)
	MemberwiseClone()	Creates a shallow copy of the current Object .(Inherited from Object .)
	Peek()	Returns the object at the beginning of the Queue without removing it.
	Synchronized(Queue)	Returns a new Queue that wraps the original queue, and is thread safe.
	ToArray()	Copies the Queue elements to a new array.
	ToString()	Returns a string that represents the current object.(Inherited from Object .)
	TrimToSize()	Sets the capacity to the actual number of elements in the Queue .

Properties

	Name	Description
	Count	Gets the number of elements contained in the Queue .
	IsSynchronized	Gets a value indicating whether access to the Queue is synchronized (thread safe).
	SyncRoot	Gets an object that can be used to synchronize access to the Queue .

Examples

C# program that uses Enqueue

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

```
class Program
{
```

```

static void Main()
{
    // New Queue of integers.
    Queue<int> q = new Queue<int>();

    q.Enqueue(5); // Add 5 to the end of the Queue.
    q.Enqueue(10); // Then add 10. 5 is at the start.
    q.Enqueue(15); // Then add 15.
    q.Enqueue(20); // Then add 20.
}
}

```

C# program that loops with Queue

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Add integers to queue.
        Queue<int> collection = new Queue<int>();
        collection.Enqueue(5);
        collection.Enqueue(6);

        // Loop through queue.
        foreach (int value in collection)
        {
            Console.WriteLine(value);
        }
    }
}

```

Output

```

5
6

```

C# program that uses Queue for request system

```

using System;
using System.Collections.Generic;

class Program
{
    enum RequestType

```

```

{
    MouseProblem,
    TextProblem,
    ScreenProblem,
    ModemProblem
};

static void Main()
{
    // Initialize help request Queue.
    Queue<RequestType> help = new Queue<RequestType>();

    // Website records a Text Problem help request.
    help.Enqueue(RequestType.TextProblem);

    // Website records a Screen Problem help request.
    help.Enqueue(RequestType.ScreenProblem);

    // You become available to take a new request.
    // ... You can't help with Mouse problems.
    if (help.Count > 0 &&
        help.Peek() != RequestType.MouseProblem)
    {
        // You solve the request.
        // ... It was a TextProblem.
        help.Dequeue();
    }

    // Record a Modem Problem help request.
    help.Enqueue(RequestType.ModemProblem);
}
}

```

C# program that copies Queue

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // New Queue of integers.
        Queue<int> queue = new Queue<int>();
        queue.Enqueue(5);
        queue.Enqueue(10);
        queue.Enqueue(15);
    }
}

```

```

queue.Enqueue(20);

// Create new array with Length equal to Queue's element count.
int[] array = new int[queue.Count];

// Copy the Queue to the int array.
queue.CopyTo(array, 0);

// Loop through and display int[] in order.
Console.WriteLine("Array:");
for (int i = 0; i < array.Length; i++)
{
    Console.WriteLine(array[i]);
}

// Loop through int array in reverse order.
Console.WriteLine("Array reverse order:");
for (int i = array.Length - 1; i >= 0; i--)
{
    Console.WriteLine(array[i]);
}
}
}

```


Output



```

Array:
5
10
15
20
Array reverse order:
20
15
10
5







```

Bit ARRAY: Methods

	Name	Description
	And(BitArray)	Performs the bitwise AND operation between the elements of the current BitArray object and the corresponding elements in the specified

		array. The current BitArray object will be modified to store the result of the bitwise AND operation.
	Clone()	Creates a shallow copy of the BitArray .
	CopyTo(Array, Int32)	Copies the entire BitArray to a compatible one-dimensional Array , starting at the specified index of the target array.
	Equals(Object)	Determines whether the specified object is equal to the current object.(Inherited from Object .)
	Get(Int32)	Gets the value of the bit at a specific position in the BitArray .
	GetEnumerator()	Returns an enumerator that iterates through the BitArray .
	GetHashCode()	Serves as the default hash function. (Inherited from Object .)
	GetType()	Gets the Type of the current instance.(Inherited from Object .)
	Not()	Inverts all the bit values in the current BitArray , so that elements set to true are changed to false, and elements set to false are changed to true.
	Or(BitArray)	Performs the bitwise OR operation between the elements of the current BitArray object and the corresponding elements in the specified array. The current BitArray object will be modified to store the result of the bitwise OR operation.
	Set(Int32, Boolean)	Sets the bit at a specific position in the BitArray to the specified value.
	SetAll(Boolean)	Sets all bits in the BitArray to the specified value.
	ToString()	Returns a string that represents the current object.(Inherited from Object .)
	Xor(BitArray)	Performs the bitwise exclusive OR operation between the elements of the current BitArray object against the corresponding elements in the specified array. The current BitArray object will be modified to store the result of the bitwise exclusive OR operation.

Properties

	Name	Description
	Count	Gets the number of elements contained in the BitArray .
	IsReadOnly	Gets a value indicating whether the BitArray is read-only.
	IsSynchronized	Gets a value indicating whether access to the BitArray is synchronized (thread safe).
	Item[Int32]	Gets or sets the value of the bit at a specific position in the BitArray .
	Length	Gets or sets the number of elements in the BitArray .
	SyncRoot	Gets an object that can be used to synchronize access to the BitArray .

Examples

C# program that creates new BitArray

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        // Create array of 5 elements and 3 true values.
        bool[] array = new bool[5];
        array[0] = true;
        array[1] = false; // <-- False value is default
        array[2] = true;
        array[3] = false;
        array[4] = true;

        // Create BitArray from the array.
        BitArray bitArray = new BitArray(array);

        // Display all bits.
        foreach (bool bit in bitArray)
        {
            Console.WriteLine(bit);
        }
    }
}

```

```
}  
}
```

Output

True
False
True
False
True

C# program that sets and counts bits

```
using System;  
using System.Collections;  
  
class Program  
{  
    static void Main()  
    {  
        // Create BitArray from the array.  
        BitArray bitArray = new BitArray(32);  
  
        // Set three bits to 1.  
        bitArray[3] = true; // You can set the bits with the indexer.  
        bitArray[5] = true;  
        bitArray.Set(10, true); // You can set the bits with Set.  
  
        // Count returns the total of all bits (1s and 0s).  
        Console.WriteLine("--- Total bits ---");  
        Console.WriteLine(bitArray.Count);  
  
        // You can loop to count set bits.  
        Console.WriteLine("--- Total bits set to 1 ---");  
        Console.WriteLine(CountBitArray(bitArray));  
    }  
  
    /// <summary>  
    /// Count set bits in BitArray.  
    /// </summary>  
    static int CountBitArray(BitArray bitArray)  
    {  
        int count = 0;  
        foreach (bool bit in bitArray)  
        {  
            if (bit)  
            {
```

```

        count++;
    }
}
return count;
}
}

```

Output

```

--- Total bits ---
32
--- Total bits set to 1 ---
3

```

C# program that displays bits and uses And

```

using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //
        // Initialize BitArray with 4 true bits and 12 false bits.
        //
        BitArray bitArray1 = new BitArray(16);
        bitArray1.Set(0, true);
        bitArray1.Set(1, true);
        bitArray1.Set(4, true);
        bitArray1.Set(5, true);

        //
        // Display the BitArray.
        //
        DisplayBitArray(bitArray1);

        //
        // Initialize BitArray with two set bits.
        //
        BitArray bitArray2 = new BitArray(16);
        bitArray2.Set(0, true);
        bitArray2.Set(7, true);
        DisplayBitArray(bitArray2);

        //
        // And the bits.
    }
}

```

```

    //
    bitArray1.And(bitArray2);
    DisplayBitArray(bitArray1);
}

/// <summary>
/// Display bits as 0s and 1s.
/// </summary>
static void DisplayBitArray(BitArray bitArray)
{
    for (int i = 0; i < bitArray.Count; i++)
    {
        bool bit = bitArray.Get(i);
        Console.Write(bit ? 1 : 0);
    }
    Console.WriteLine();
}
}

```

Output

```

110011000000000000
100000010000000000
100000000000000000

```

Lab No 5:

Generics

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type.

1. List :

The generic version of ArrayList is List. It uses the same functions/ Methods and properties as ArrayList uses.

Syntax:

```
List <datatype> obj = new List<datatype> ();
```

2. Dictionary :

The generic version of Hashtable is Dictionary. It uses the same functions/Methods and properties as Hashtable uses.

Syntax:

```
Dictionary <datatype Key, datatype Value> obj = new Dictionary < datatype Key,  
datatype Value > ();
```

Lab No 6:

Overriding

Override, in C#, is a keyword used to replace a virtual member that is defined in a base class with the definition of that member in the derived class.

The override modifier allows programmers to specify the specialization of an existing virtual member inherited from a base class to provide a new implementation of that member in the derived class. It can be used with a method, property, indexer or an event that needs to be modified or extended in a derived class.

The override modifier is intended to implement the concept of polymorphism in C#.

Override differs from new modifiers in that the former is used only to override a virtual member of a base class while the latter also helps to override a non-virtual member defined in a base class by hiding the definition contained in the base class

Task for Overriding Functions:

- **Add (Object o)** “The Original Function takes an object as an argument and allow duplication in addition”
 - ➔ You have to Override this Add function and change its functionality that it does not allow the duplication in an ArrayList.
- **Remove (Object o)** “The Original Function takes an object as an argument and Removes the first occurrence of that object from ArrayList”

- ➔ You have to Override this Remove function and change its functionality that it removes the entire Occurrence from the ArrayList.
- **Insert (int index , Object o)** “The Original Function takes index no. and an object as an argument and Insert the value at that index but it does not Overwrite the value”
 - ➔ You have to Override this Insert function and change its functionality that it will overwrite the value at that location that the other value at that location will be overwrite by the new object given in the parameter.
- **Sort ()** “The Original Function Sorts the entire list in Ascending order”
 - ➔ You have to Override this Sort function and change its functionality that it sorts the ArrayList in Descending order.

Lab No 7:

LINQ (Language-Integrated Query)

The acronym LINQ stands for Language Integrated Query. Microsoft’s query language is fully integrated and offers easy data access from in-memory objects, databases, XML documents, and many more. It is through a set of extensions, LINQ ably integrates queries in C# and Visual Basic. This tutorial offers a complete insight into LINQ with ample examples and coding. The entire tutorial is divided into various topics with subtopics that a beginner can be able to move gradually to more complex topics of LINQ.

Advantages of LINQ

LINQ offers a host of advantages and among them the foremost is its powerful expressiveness which enables developers to express declaratively. Some of the other advantages of LINQ are given below.

- LINQ offers syntax highlighting that proves helpful to find out mistakes during design time.
- LINQ offers IntelliSense which means writing more accurate queries easily.
- Writing codes is quite faster in LINQ and thus development time also gets reduced significantly.

- LINQ makes easy debugging due to its integration in the C# language.
- Viewing relationship between two tables is easy with LINQ due to its hierarchical feature and this enables composing queries joining multiple tables in less time.
- LINQ allows usage of a single LINQ syntax while querying many diverse data sources and this is mainly because of its unitive foundation.
- LINQ is extensible that means it is possible to use knowledge of LINQ to querying new data source types.
- LINQ offers the facility of joining several data sources in a single query as well as breaking complex problems into a set of short queries easy to debug.
- LINQ offers easy transformation for conversion of one data type to another like transforming SQL data to XML data.

Need For LINQ

Prior to LINQ, it was essential to learn C#, SQL, and various APIs that bind together the both to form a complete application. Since, these data sources and programming languages face an impedance mismatch; a need of short coding is felt.

Below is an example of how many diverse techniques were used by the developers while querying a data before the advent of LINQ.

```
SqlConnection sqlConnection = new SqlConnection(connectString);
SqlConnection.Open();

System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;

sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader (CommandBehavior.CloseConnection)
```

Interestingly, out of the featured code lines, query gets defined only by the last two. Using LINQ, the same data query can be written in a readable color-coded form like the following one mentioned below that too in a very less time.

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");
```

```
var query = from c in db.Customers select c;
```

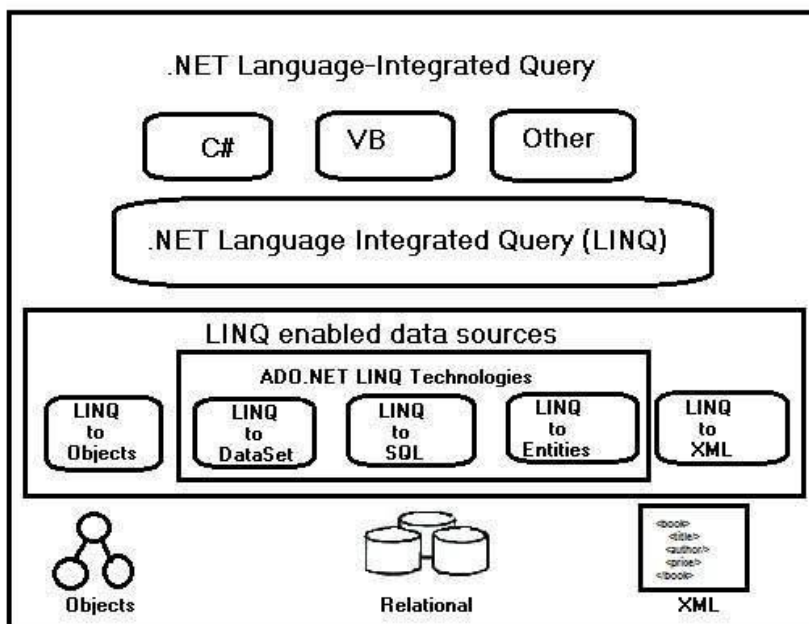
Types of LINQ

The types of LINQ are mentioned below in brief.

- LINQ to Objects
- LINQ to XML(XLINQ)
- LINQ to DataSet
- LINQ to SQL (DLINQ)
- LINQ to Entities

LINQ Architecture in .NET

LINQ has a 3-layered architecture in which the uppermost layer consists of the language extensions and the bottom layer consists of data sources that are typically objects implementing `IEnumerable <T>` or `IQueryable <T>` generic interfaces. The architecture is shown below.



Difference between LINQ and Stored Procedure

There is an array of differences existing between LINQ and Stored procedures. These differences are mentioned below.

- Stored procedures are much faster than a LINQ query as they follow an expected execution plan.
- It is easy to avoid run-time errors while executing a LINQ query than in comparison to a stored procedure as the former has Visual Studio's Intellisense support as well as full-type checking during compile-time.
- LINQ allows debugging by making use of .NET debugger which is not in case of stored procedures.
- LINQ offers support for multiple databases in contrast to stored procedures, where it is essential to re-write the code for diverse types of databases.
- Deployment of LINQ based solution is easy and simple in comparison to deployment of a set of stored procedures.

Lab No 8:

LINQ - Query Operators

A set of extension methods forming a query pattern is known as LINQ Standard Query Operators. As building blocks of LINQ query expressions, these operators offer a range of query capabilities like filtering, sorting, projection, aggregation, etc.

LINQ standard query operators can be categorized into the following ones on the basis of their functionality.

- Filtering Operators
- Join Operators
- Projection Operations
- Sorting Operators
- Grouping Operators
- Conversions
- Concatenation
- Aggregation
- Quantifier Operations
- Partition Operations
- Generation Operations

- Set Operations
- Equality
- Element Operators

Filtering Operators

Filtering is an operation to restrict the result set such that it has only selected elements satisfying a particular condition.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
where	Filter values based on a predicate function	where	Where
OfType	Filter values based on their ability to be as a specified type	Not Applicable	Not Applicable

Join Operators

Joining refers to an operation in which data sources with difficult to follow relationships with each other in a direct way are targeted.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Join	The operator join two sequences on basis of matching keys	join ... in ... on ... equals ...	From x In ..., y In ... Where x.a = y.a
GroupJoin	Join two sequences and group the matching elements	join ... in ... on ... equals ... into ...	Group Join ... In ... On ...

Projection Operations

Projection is an operation in which an object is transformed into an altogether new form with only specific properties.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Select	The operator projects values on basis of a transform function	select	Select
SelectMany	The operator project the sequences of values which are based on a transform function as well as flattens them into a single sequence	Use multiple from clauses	Use multiple From clauses

Sorting Operators

A sorting operation allows ordering the elements of a sequence on basis of a single or more attributes.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
OrderBy	The operator sort values in an ascending order	orderby	Order By
OrderByDescending	The operator sort values in a descending order	orderby ... descending	Order By ... Descending
ThenBy	Executes a secondary sorting in an ascending order	orderby ..., ...	Order By ..., ...
ThenByDescending	Executes a secondary sorting in a descending order	orderby ..., ... descending	Order By ..., ... Descending

Reverse	Performs a reversal of the order of the elements in a collection	Not Applicable	Not Applicable
---------	------------------------------------------------------------------	----------------	----------------

Grouping Operators

The operators put data into some groups based on a common shared attribute.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
GroupBy	Organize a sequence of items in groups and return them as an IEnumerable collection of type IGrouping<key, element>	group ... by -or- group ... by ... into ...	Group ... By ... Into ...
ToLookup	Execute a grouping operation in which a sequence of key pairs are returned	Not Applicable	Not Applicable

Conversions

The operators change the type of input objects and are used in a diverse range of applications.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
AsEnumerable	Returns the input typed as IEnumerable<T>	Not Applicable	Not Applicable
AsQueryable	A (generic) IEnumerable is converted to a (generic) IQueryable	Not Applicable	Not Applicable
Cast	Performs casting of elements of a collection to a specified	Use an explicitly typed range variable. Eg:from	From ... As ...

	type	string str in words	
OfType	Filters values on basis of their , depending on their capability to be cast to a particular type	Not Applicable	Not Applicable
ToArray	Forces query execution and does conversion of a collection to an array	Not Applicable	Not Applicable
ToDictionary	On basis of a key selector function set elements into a Dictionary<TKey, TValue> and forces execution of a LINQ query	Not Applicable	Not Applicable
ToList	Forces execution of a query by converting a collection to a List<T>	Not Applicable	Not Applicable
ToLookup	Forces execution of a query and put elements into a Lookup<TKey, TElement> on basis of a key selector function	Not Applicable	Not Applicable

Concatenation

Performs concatenation of two sequences and is quite similar to the Union operator in terms of its operation except of the fact that this does not remove duplicates.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Concat	Two sequences are concatenated for the formation of a single one sequence.	Not Applicable	Not Applicable

Aggregation

Performs any type of desired aggregation and allows creating custom aggregations in LINQ.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Aggregate	Operates on the values of a collection to perform custom aggregation operation	Not Applicable	Not Applicable
Average	Average value of a collection of values is calculated	Not Applicable	Aggregate ... In ... Into Average()
Count	Counts the elements satisfying a predicate function within collection	Not Applicable	Aggregate ... In ... Into Count()
LongCount	Counts the elements satisfying a predicate function within a huge collection	Not Applicable	Aggregate ... In ... Into LongCount()
Max	Find out the maximum value within a collection	Not Applicable	Aggregate ... In ... Into Max()
Min	Find out the minimum value existing within a collection	Not Applicable	Aggregate ... In ... Into Min()
Sum	Find out the sum of a values within a collection	Not Applicable	Aggregate ... In ... Into Sum()

Quantifier Operations

These operators return a Boolean value i.e. True or False when some or all elements within a sequence satisfy a specific condition.

Show Examples

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
All	Returns a value 'True' if all elements of a sequence satisfy a predicate condition	Not Applicable	Aggregate ... In ... Into All(...)
Any	Determines by searching a sequence that whether any element of the same satisfy a specified condition	Not Applicable	Aggregate ... In ... Into Any()
Contains	Returns a 'True' value if finds that a specific element is there in a sequence if the sequence does not contain that specific element, 'false' value is returned	Not Applicable	Not Applicable

Partition Operators

Divide an input sequence into two separate sections without rearranging the elements of the sequence and then returning one of them.

Show Examples

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Skip	Skips some specified number of elements within a sequence and returns the remaining ones	Not Applicable	Skip
SkipWhile	Same as that of Skip with the only exception that number of elements to skip are specified by a Boolean condition	Not Applicable	Skip While
Take	Take a specified number of elements from a sequence and skip the remaining ones	Not Applicable	Take

TakeWhile	Same as that of Take except the fact that number of elements to take are specified by a Boolean condition	Not Applicable	Take While
-----------	-----------------------------------------------------------------------------------------------------------	----------------	------------

Generation Operations

A new sequence of values is created by generational operators.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
DefaultIfEmpty	When applied to an empty sequence, generate a default element within a sequence	Not Applicable	Not Applicable
Empty	Returns an empty sequence of values and is the most simplest generational operator	Not Applicable	Not Applicable
Range	Generates a collection having a sequence of integers or numbers	Not Applicable	Not Applicable
Repeat	Generates a sequence containing repeated values of a specific length	Not Applicable	Not Applicable

Set Operations

There are four operators for the set operations, each yielding a result based on different criteria.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Distinct	Results a list of unique values from a collection by filtering duplicate data if any	Not Applicable	Distinct

Except	Compares the values of two collections and return the ones from one collection who are not in the other collection	Not Applicable	Not Applicable
Intersect	Returns the set of values found to be identical in two separate collections	Not Applicable	Not Applicable
Union	Combines content of two different collections into a single list that too without any duplicate content	Not Applicable	Not Applicable

Equality

Compares two sentences (enumerable) and determine are they an exact match or not.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
SequenceEqual	Results a Boolean value if two sequences are found to be identical to each other	Not Applicable	Not Applicable

Element Operators

Except the DefaultIfEmpty, all the rest eight standard query element operators return a single element from a collection.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
ElementAt	Returns an element present within a specific index in a collection	Not Applicable	Not Applicable
ElementAtOrDefault	Same as ElementAt except of the fact that it also returns a default value in case the	Not Applicable	Not Applicable

	specific index is out of range		
First	Retrieves the first element within a collection or the first element satisfying a specific condition	Not Applicable	Not Applicable
FirstOrDefault	Same as First except the fact that it also returns a default value in case there is no existence of such elements	Not Applicable	Not Applicable
Last	Retrieves the last element present in a collection or the last element satisfying a specific condition	Not Applicable	Not Applicable
LastOrDefault	Same as Last except the fact that it also returns a default value in case there is no existence of any such element	Not Applicable	Not Applicable
Single	Returns the lone element of a collection or the lone element that satisfy a certain condition	Not Applicable	Not Applicable
SingleOrDefault	Same as Single except that it also returns a default value if there is no existence of any such lone element	Not Applicable	Not Applicable
DefaultIfEmpty	Returns a default value if the collection or list is empty or null	Not Applicable	Not Applicable

Filtering Operators - Where

Filtering operators in LINQ filter the sequence (collection) based on some given criteria.

The following table lists all the filtering operators available in LINQ.

Filtering Operators	Description
Where	Returns values from the collection based on a predicate function
OfType	Returns values from the collection based on a specified type. However, it will depend on their ability to cast to a specified type.

Where

The Where operator (Linq extension method) filters the collection based on a given criteria expression and returns a new collection. The criteria can be specified as lambda expression or Func delegate type.

The **Where** extension method has following two overloads. Both overload methods accepts a [Func delegate](#) type parameter. One overload required `Func<TSource, bool>` input parameter and second overload method required `Func<TSource, int, bool>` input parameter where int is for index:

Where() method overloads:

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                Func<TSource, bool> predicate);
```

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                Func<TSource, int, bool> predicate);
```

Where clause in Query Syntax:

The following query sample uses a Where operator to filter the students who is teen ager from the given collection (sequence). It uses a lambda expression as a predicate function.

Example: Where clause - LINQ query syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
};

var filteredResult = from s in studentList
                    where s.Age > 12 && s.Age < 20
                    select s.StudentName;
```

Example: Where clause - LINQ query syntax in VB.Net

```
Dim studentList = New List(Of Student) From {  
    New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},  
    New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},  
    New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},  
    New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},  
    New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}  
}  
  
Dim filteredResult = From s In studentList  
    Where s.Age > 12 And s.Age < 20  
    Select s.StudentName
```

In the above example, filteredResult will include following students after query execution.



In the above sample query, the lambda expression body `s.Age > 12 && s.Age < 20` is passed as a predicate function `Func<TSource, bool>` that evaluates every student in the collection.

Alternatively, you can also use a Func type delegate with an anonymous method to pass as a predicate function as below (output would be the same):

Exapmle: Where clause

```
Func<Student,bool> isTeenAger = delegate(Student s) {  
    return s.Age > 12 && s.Age < 20;  
};  
  
var filteredResult = from s in studentList  
    where isTeenAger(s)  
    select s;
```

You can also call any method that matches with Func parameter with one of Where() method overloads.

Exapmle: Where clause

```
public static void Main()  
{  
    var filteredResult = from s in studentList  
        where isTeenAger(s)
```

```

        select s;
    }

    public static bool IsTeenAger(Student stud)
    {
        return stud.Age > 12 && stud.Age < 20;
    }

```

Where extension method in Method Syntax:

Unlike the query syntax, you need to pass whole lambda expression as a predicate function instead of just body expression in LINQ method syntax.

Example: Where in method syntax in C#

```
var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

Example: Where in method syntax in VB.Net

```
Dim filteredResult = studentList.Where(Function(s) s.Age > 12 And s.Age < 20 )
```

As mentioned above, the **Where** extension method also have second overload that includes index of current element in the collection. You can use that index in your logic if you need.

The following example uses the Where clause to filter out odd elements in the collection and return only even elements. Please remember that index starts from zero.

Example: Linq - Where extension method in C#

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 19 }
};

var filteredResult = studentList.Where((s, i) => {
    if(i % 2 == 0) // if it is even element
        return true;

    return false;
});

foreach (var std in filteredResult)
    Console.WriteLine(std.StudentName);

```

Output:



Multiple Where clause:

You can call the Where() extension method more than one time in a single LINQ query.

Example: Multiple where clause in Query Syntax C#

```
var filteredResult = from s in studentList
                     where s.Age > 12
                     where s.Age < 20
                     select s;
```

Example: Multiple where clause in Method Syntax C#

```
var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);
```

Points to Remember :

1. **Where** is used for filtering the collection based on given criteria.
2. Where extension method has two overload methods. Use a second overload method to know the index of current element in the collection.
3. Method Syntax requires the whole lambda expression in Where extension method whereas Query syntax requires only expression body.
4. Multiple **Where** extension methods are valid in a single LINQ query.

Sorting Operators: OrderBy & OrderByDescending

A sorting operator arranges the elements of the collection in ascending or descending order. LINQ includes following sorting operators.

Sorting Operator	Description
OrderBy	Sorts the elements in the collection based on specified fields in ascending or descending order.
OrderByDescending	Sorts the collection based on specified fields in descending order. Only valid in

Sorting Operator	Description
	method syntax.
ThenBy	Only valid in method syntax. Used for second level sorting in ascending order.
ThenByDescending	Only valid in method syntax. Used for second level sorting in descending order.
Reverse	Only valid in method syntax. Sorts the collection in reverse order.

OrderBy:

OrderBy sorts the values of a collection in ascending or descending order. It sorts the collection in ascending order by default because **ascending** keyword is optional here. Use descending keyword to sort collection in descending order.

Example: OrderBy in Query Syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var orderByResult = from s in studentList
    orderby s.StudentName
    select s;

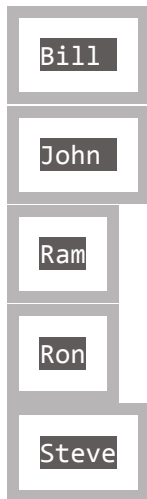
var orderByDescendingResult = from s in studentList
    orderby s.StudentName descending
    select s;
```

Example: OrderBy in Query Syntax VB.Net

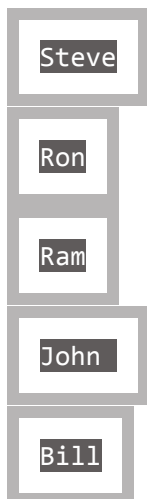
```
Dim orderByResult = From s In studentList
    Order By s.StudentName
    Select s

Dim orderByDescendingResult = From s In studentList
    Order By s.StudentName Descending
    Select s
```

orderByResult in the above example would contain following elements after execution:



orderByDescendingResult in the above example would contain following elements after execution:



OrderBy in Method Syntax:

OrderBy extension method has two overloads. First overload of OrderBy extension method accepts the Func delegate type parameter. So you need to pass the lambda expression for the field based on which you want to sort the collection.

The second overload method of OrderBy accepts object of IComparer along with Func delegate type to use custom comparison for sorting.

OrderBy() method overloads:

```
public static IObservable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource>
source,
                                                                    Func<TSource, TKey>
keySelector);

public static IObservable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource>
source,
                                                                    Func<TSource, TKey>
keySelector,
                                                                    IComparer<TKey>
comparer);
```


The following example sorts the studentList collection in ascending order of StudentName using OrderBy extension method.

Example: OrderBy in Method Syntax C#

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};  
  
var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);
```

Example: OrderBy in Method Syntax VB.Net

```
Dim studentsInAscOrder = studentList.OrderBy(Function(s) s.StudentName)
```

Note : Method syntax does not allow the decending keyword to sorts the collection in decending order. Use OrderByDecending() method for it.

OrderByDescending:

OrderByDescending sorts the collection in descending order.

OrderByDescending is valid only with the Method syntax. It is not valid in query syntax because the query syntax uses ascending and descending attributes as shown above.

Example: OrderByDescending C#

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};  
  
var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);
```

Example: OrderByDescending VB.Net

```
Dim studentsInDescOrder = studentList.OrderByDescending(Function(s) s.StudentName)
```

A result in the above example would contain following elements after execution.



Please note that OrderByDescending is not supported in query syntax. Use the descending keyword instead.

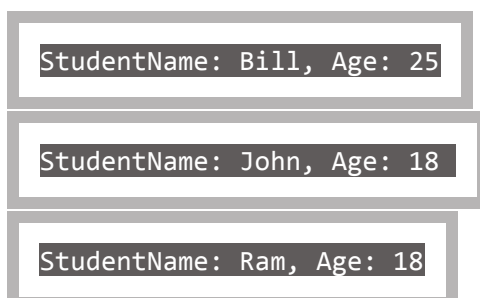
Multiple Sorting:

You can sort the collection on multiple fields separated by comma. The given collection would be first sorted based on the first field and then if value of first field would be the same for two elements then it would use second field for sorting and so on.

Example: Multiple sorting in Query syntax C#

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 } ,  
    new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }  
};  
  
var orderByResult = from s in studentList  
                    orderby s.StudentName, s.Age  
                    select new { s.StudentName, s.Age };
```

In the above example, studentList collection includes two identical StudentNames, Ram. So now, studentList would be first sorted based on StudentName and then by Age in ascending order. So, orderByResult would contain following elements after execution



StudentName: Ram, Age: 20

StudentName: Ron, Age: 19

StudentName: Steve, Age: 15

Note : Multiple sorting in method syntax works differently. Use ThenBy or ThenByDescending extension methods for secondary sorting.

Points to Remember:

1. LINQ includes five sorting operators: OrderBy, OrderByDescending, ThenBy, ThenByDescending and Reverse
2. LINQ query syntax does not support OrderByDescending, ThenBy, ThenByDescending and Reverse. It only supports 'Order By' clause with 'ascending' and 'descending' sorting direction.
3. LINQ query syntax supports multiple sorting fields separated by comma whereas you have to use ThenBy & ThenByDescending methods for secondary sorting.

Sorting Operators: ThenBy & ThenByDescending

We have seen how to do sorting using multiple fields in query syntax in the previous section.

Multiple sorting in method syntax is supported by using ThenBy and ThenByDescending extension methods.

The OrderBy() method sorts the collection in ascending order based on specified field. Use ThenBy() method after OrderBy to sort the collection on another field in ascending order. Linq will first sort the collection based on primary field which is specified by OrderBy method and then sort the resulted collection in ascending order again based on secondary field specified by ThenBy method.

The same way, use ThenByDescending method to apply secondary sorting in descending order.

The following example shows how to use ThenBy and ThenByDescending method for second level sorting:

Example: ThenBy & ThenByDescending

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
```

```

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 },
new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
new Student() { StudentID = 4, StudentName = "Ram", Age = 20 },
new Student() { StudentID = 5, StudentName = "Ron", Age = 19 },
new Student() { StudentID = 6, StudentName = "Ram", Age = 18 }
};
var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s =>
s.Age);

```

As you can see in the above example, we first sort a studentList collection by StudentName and then by Age. So now, thenByResult would contain following elements after sorting:

StudentName: Bill, Age: 25
StudentName: John, Age: 18
StudentName: Ram, Age: 18
StudentName: Ram, Age: 20
StudentName: Ron, Age: 19
StudentName: Steve, Age: 15

thenByDescResult would contain following elements. Please notice that Ram with age 20 comes before Ram with age 18 because it has used ThenByDescending.

StudentName: Bill, Age: 25
StudentName: John, Age: 18
StudentName: Ram, Age: 20
StudentName: Ram, Age: 18
StudentName: Ron, Age: 19
StudentName: Steve, Age: 15

You can use ThenBy and ThenByDescending method same way in VB.Net as below:

Example: ThenBy & ThenByDescending VB.Net

```
Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)
                                .ThenBy(Function(s) s.Age)

Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)
                                .ThenByDescending(Function(s) s.Age)
```

Points to Remember :

1. OrderBy and ThenBy sorts collections in ascending order by default.
2. ThenBy or ThenByDescending is used for second level sorting in method syntax.
3. ThenByDescending method sorts the collection in descending order on another field.
4. ThenBy or ThenByDescending is NOT applicable in Query syntax.
5. Apply secondary sorting in query syntax by separating fields using comma.

Grouping Operators: GroupBy & ToLookup

The grouping operators do the same thing as the GroupBy clause of SQL query. The grouping operators create a group of elements based on the given key. This group is contained in a special type of collection that implements an `IGrouping<TKey,TSource>` interface where `TKey` is a key value, on which the group has been formed and `TSource` is the collection of elements that matches with the grouping key value.

Grouping Operators	Description
GroupBy	The GroupBy operator returns groups of elements based on some key value. Each group is represented by <code>IGrouping<TKey, TElement></code> object.
ToLookup	ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate.

GroupBy:

A LINQ query can end with a GroupBy or Select clause.

The GroupBy operator returns a group of elements from the given collection based on some key value. Each group is represented by `IGrouping<TKey, TElement>` object. Also, the GroupBy method has eight overload methods, so you can use appropriate extension method based on your requirement in method syntax.

The result of GroupBy operators is a collection of groups. For example, GroupBy returns IEnumerable<IGrouping<TKey,Student>> from the Student collection:

```
var groupedResult = studentList.GroupBy()
```

▲ 1 of 8 ▼ (extension) **IEnumerable<IGrouping<TKey,Student>>** IEnumerable<Student>.GroupBy(Func<Student,TKey> keySelector)
Groups the elements of a sequence according to a specified key selector function.
keySelector: A function to extract the key for each element.

Return

type of GroupBy()

GroupBy in Query Syntax:

The following example creates a groups of students who have same age. Students of the same age will be in the same collection and each grouped collection will have a key and inner collection, where the key will be the age and the inner collection will include students whose age is matched with a key.

Example: GroupBy in Query syntax C#

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 21 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,  
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Abram", Age = 21 }  
};  
  
var groupedResult = from s in studentList  
                    group s by s.Age;  
  
//iterate each group  
foreach (var ageGroup in groupedResult)  
{  
    Console.WriteLine("Age Group: {0}", ageGroup .Key); //Each group has a key  
  
    foreach(Student s in ageGroup) // Each group has inner collection  
        Console.WriteLine("Student Name: {0}", s.StudentName);  
}
```

Output:

AgeGroup: 18

StudentName: John

StudentName: Bill

AgeGroup: 21

StudentName: Steve

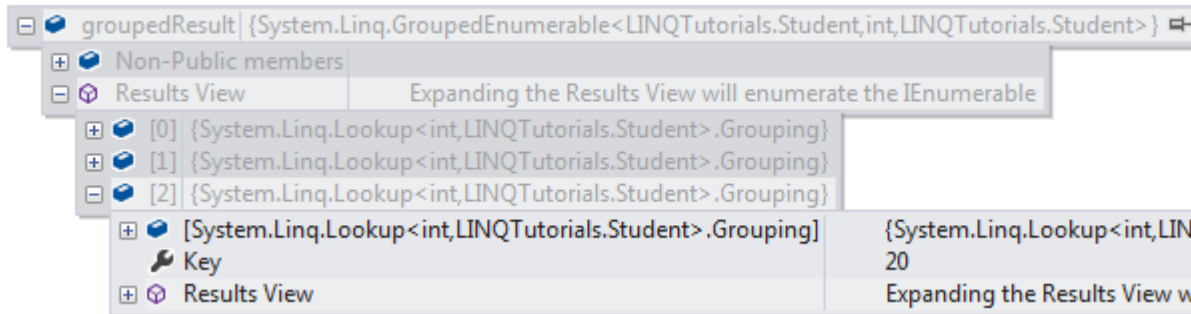
StudentName: Abram

AgeGroup: 20

StudentName: Ram

As you can see in the above example, you can iterate the group using a 'foreach' loop, where each group contains a key and inner collection. The following figure shows the result in debug view.

```
var groupedResult = from s in studentList
```



Grouped collection with key and inner collection

Use "Into Group" with the 'Group By' clause in VB.Net as shown below.

Example: GroupBy clause in VB.Net

```
Dim groupQuery = From s In studentList
                  Group By s.Age Into Group

For Each group In groupQuery
    Console.WriteLine("Age Group: {0}", group.Age) // Each group has key property name

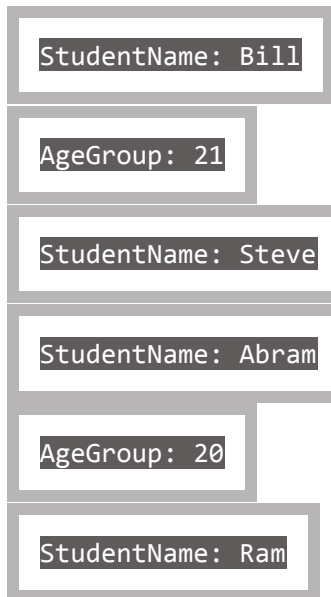
    For Each student In group.Group // Each group has inner collection
        Console.WriteLine("Student Name: {0}", student.StudentName)
    Next
Next
```

Notice that each group will have a property name on which group is performed. In the above example, we have used Age to form a group so each group will have "Age" property name instead of "Key" as a property name.

Output:

AgeGroup: 18

StudentName: John



GroupBy in Method Syntax:

The GroupBy() extension method works the same way in the method syntax. Specify the lambda expression for key selector field name in GroupBy extension method.

Example: GroupBy in method syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Abram", Age = 21 }
};

var groupedResult = studentList.GroupBy(s => s.Age);

foreach (var ageGroup in groupedResult)
{
    Console.WriteLine("Age Group: {0}", ageGroup.Key); //Each group has a key

    foreach(Student s in ageGroup) //Each group has a inner collection
        Console.WriteLine("Student Name: {0}", s.StudentName);
}
```

Example: GroupBy in method syntax VB.Net

```
Dim groupQuery = studentList.GroupBy(Function(s) s.Age)

For Each ageGroup In groupQuery

    Console.WriteLine("Age Group: {0}", ageGroup.Key) //Each group has a key

    For Each student In ageGroup.AsEnumerable() //Each group has a inner collection
        Console.WriteLine("Student Name: {0}", student.StudentName)
    Next
Next
```


Output:

AgeGroup: 18

StudentName: John

StudentName: Bill

AgeGroup: 21

StudentName: Steve

StudentName: Abram

AgeGroup: 20

StudentName: Ram

ToLookup

ToLookup is the same as GroupBy; the only difference is GroupBy execution is deferred, whereas ToLookup execution is immediate. Also, ToLookup is only applicable in Method syntax. **ToLookup is not supported in the query syntax.**

Example: ToLookup in method syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Abram", Age = 21 }
};

var lookupResult = studentList.ToLookup(s => s.age);

foreach (var group in lookupResult)
{
    Console.WriteLine("Age Group: {0}", group.Key); //Each group has a key

    foreach(Student s in group) //Each group has a inner collection
        Console.WriteLine("Student Name: {0}", s.StudentName);
}
```

Example: ToLookup in method syntax VB.Net

```
Dim loopupResult = studentList.ToLookup(Function(s) s.Age)
```

Points to Remember :

1. GroupBy & ToLookup return a collection that has a key and an inner collection based on a key field value.
2. The execution of GroupBy is deferred whereas that of ToLookup is immediate.
3. A LINQ query syntax can be end with the GroupBy or Select clause.

Joining Operator: Join

The joining operators joins the two sequences (collections) and produce a result.

Joining Operators	Usage
Join	The Join operator joins two sequences (collections) based on a key and returns a resulted sequence.
GroupJoin	The GroupJoin operator joins two sequences based on keys and returns groups of sequences. It is like Left Outer Join of SQL.

Join:

The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression. It is the same as **inner join** of SQL.

Join in Method Syntax:

The Join extension method has two overloads as shown below.

Join() method overloads:

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this
IEnumerable<TOuter> outer,
IEnumerable<TInner> inner,
Func<TOuter, TKey> outerKeySelector,
Func<TInner, TKey>
innerKeySelector,
Func<TOuter, TInner, TResult>
resultSelector);
```

```

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this
IEnumerable<TOuter> outer,
                                                                    IEnumerable<TInner> inner,
                                                                    Func<TOuter, TKey>
outerKeySelector,
                                                                    Func<TInner, TKey>
innerKeySelector,
                                                                    Func<TOuter, TInner, TResult>
resultSelector,
                                                                    IEqualityComparer<TKey> comparer);

```

As you can see in the first overload method takes five input parameters (except the first 'this' parameter): 1) outer 2) inner 3) outerKeySelector 4) innerKeySelector 5) resultSelector.

Let's take a simple example. The following example joins two string collection and return new collection that includes matching strings in both the collection.

Example: Join operator C#

```

IList<string> strList1 = new List<string>() {
    "One",
    "Two",
    "Three",
    "Four"
};

IList<string> strList2 = new List<string>() {
    "One",
    "Two",
    "Five",
    "Six"
};

var innerJoin = strList1.Join(strList2,
    str1 => str1,
    str2 => str2,
    (str1, str2) => str1);

```



Now, let's understand join method using following Student and Standard class where Student class includes StandardID that matches with StandardID of Standard class.

Example Classes

```

public class Student{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int StandardID { get; set; }
}

```

```

}

public class Standard{
    public int StandardID { get; set; }
    public string StandardName { get; set; }
}

```

The following example demonstrates LINQ Join query.

Example: Join Query C#

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", StandardID =1 },
    new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },
    new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },
    new Student() { StudentID = 4, StudentName = "Ram" , StandardID =2 },
    new Student() { StudentID = 5, StudentName = "Ron" }
};

IList<Standard> standardList = new List<Standard>() {
    new Standard(){ StandardID = 1, StandardName="Standard 1"},
    new Standard(){ StandardID = 2, StandardName="Standard 2"},
    new Standard(){ StandardID = 3, StandardName="Standard 3"}
};

var innerJoin = studentList.Join(// outer sequence
                                standardList, // inner sequence
                                student => student.StandardID, // outerKeySelector
                                standard => standard.StandardID, // innerKeySelector
                                (student, standard) => new // result selector
                                {
                                    StudentName = student.StudentName,
                                    StandardName = standard.StandardName
                                });

```

The following image illustrate the parts of Join operator in the above example.

```

var joinResult = studentList.Join(standardList,
    student => student.StandardID,
    standard => standard.StandardID,
    (student, standard) => new
    {
        StudentFullName = student.StudentName,
        StandarFullIdName = standard.StandardName
    });

```

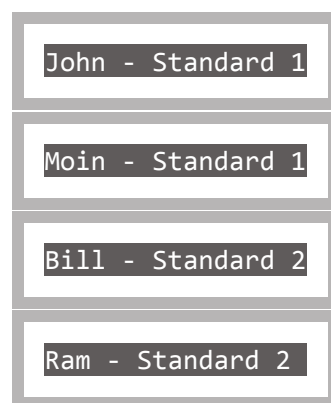
join operator

In the above example of join query, studentList is outer sequence because query starts from it. First parameter in Join method is used to specify the inner sequence which is standardList in the above example. Second and third parameter of Join method is used to specify a field whose value should be match using lambda expression in order to include element in the result. The key selector for the outer

sequence `student => student.StandardID` indicates that take StandardID field of each elements of studentList should be match with the key of inner sequence `standard => standard.StandardID`. If value of both the key field is matched then include that element into result.

The last parameter in Join method is an expression to formulate the result. In the above example, result selector includes StudentName and StandardName property of both the sequence.

StandardID Key of both the sequences (collections) must match otherwise the item will not be included in the result. For example, Ron is not associated with any standard so Ron is not included in the result collection. innerJoinResult in the above example would contain following elements after execution:



The following example demonstrates the Join operator in method syntax in VB.Net.

Example: Join operator VB.Net

```
Dim innerJoin = studentList.Join(standardList,
    Function(s) s.StandardID,
    Function(std) std.StandardID,
    Function(s, std) New With
    {
        .StudentName = s.StudentName,
        .StandardName = std.StandardName
    });
```

Join in Query Syntax:

Join operator in query syntax works slightly different than method syntax. It requires outer sequence, inner sequence, key selector and result selector. 'on' keyword is used for key selector where left side of 'equals' operator is outerKeySelector and right side of 'equals' is innerKeySelector.

Syntax: Join in query syntax

```

from ... in outerSequence

join ... in innerSequence

on outerKey equals innerKey

select ...

```

The following example of Join operator in query syntax returns a collection of elements from studentList and standardList if their Student.StandardID and Standard.StandardID is match.

Example: Join operator in query syntax C#

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13, StandardID =1 },
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21, StandardID =1 },
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18, StandardID =2 },
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
};

IList<Standard> standardList = new List<Standard>() {
    new Standard(){ StandardID = 1, StandardName="Standard 1"},
    new Standard(){ StandardID = 2, StandardName="Standard 2"},
    new Standard(){ StandardID = 3, StandardName="Standard 3"}
};

var innerJoin = from s in studentList // outer sequence
                join st in standardList //inner sequence
                on s.StandardID equals st.StandardID // key selector
                select new { // result selector
                    StudentName = s.StudentName,
                    StandardName = st.StandardName
                };

```

Example: Join operator in query syntax VB.Net

```

Dim innerJoin = From s In studentList ' outer sequence
                Join std In standardList ' inner sequence
                On s.StandardID Equals std.StandardID ' key selector
                Select _ ' result selector
                    StudentName = s.StudentName,
                    StandardName = std.StandardName

```

Output:

John - Standard 1

Moin - Standard 1

Bill - Standard 2

Ram - Standard 2

Note : Use the **equals** operator to match key selector in query syntax. **==** is not valid.

Points to Remember :

1. **Join** and **GroupJoin** are joining operators.
2. **Join** is like inner join of SQL. It returns a new collection that contains common elements from two collections whose keys match.
3. **Join** operates on two sequences inner sequence and outer sequence and produces a result sequence.
4. **Join** query

Projection Operators: Select, SelectMany

There are two projection operators available in LINQ. 1) Select 2) SelectMany

Select:

The Select operator always returns an IEnumerable collection which contains elements based on a transformation function. It is similar to the Select clause of SQL that produces a flat result set.

Now, let's understand Select query operator using the following Student class.

Example Classes

```
public class Student{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

Select in Query Syntax:

LINQ query syntax must end with a **Select** or **GroupBy** clause. The following example demonstrates select operator that returns a string collection of StudentName.

Example: Select in query syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John" },
    new Student() { StudentID = 2, StudentName = "Moin" },
    new Student() { StudentID = 3, StudentName = "Bill" },
    new Student() { StudentID = 4, StudentName = "Ram" },
    new Student() { StudentID = 5, StudentName = "Ron" }
```

```
};

var selectResult = from s in studentList
                   select s.StudentName;
```

The select operator can be used to format the result as per our requirement. It can be used to return a collection of custom class or anonymous type which includes properties as per our need.

The following example of the select clause returns a collection of [anonymous type](#) containing the Name and Age property.

Example: Select operator in query syntax C#

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 },
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 },
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 },
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 },
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};

// returns collection of anonymous objects with Name and Age property
var selectResult = from s in studentList
                   select new { Name = "Mr. " + s.StudentName, Age = s.Age };

// iterate selectResult
foreach (var item in selectResult)
    Console.WriteLine("Student Name: {0}, Age: {1}", item.Name, item.Age);
```

Example: Select operator in query syntax VB.Net

```
Dim selectResult = From s In studentList
                   Select New With {.Name = s.StudentName, .Age = s.Age}
```

Output:

```
Student Name: Mr. John, Age: 13
```

```
Student Name: Mr. Moin, Age: 21
```

```
Student Name: Mr. Bill, Age: 18
```

```
Student Name: Mr. Ram, Age: 20
```

```
Student Name: Mr. Ron, Age: 15
```


Select in Method Syntax:

The Select operator is optional in method syntax. However, you can use it to shape the data. In the following example, Select extension method returns a collection of anonymous object with the Name and Age property:

Example: Select in method syntax C#

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }  
};  
  
var selectResult = studentList.Select(s => new { Name = s.StudentName ,  
                                                Age = s.Age  });
```

Aggregation Operators: Aggregate

The aggregation operators perform mathematical operations like Average, Aggregate, Count, Max, Min and Sum, on the numeric property of the elements in the collection.

Method	Description
Aggregate	Performs a custom aggregation operation on the values in the collection.
Average	calculates the average of the numeric items in the collection.
Count	Counts the elements in a collection.
LongCount	Counts the elements in a collection.
Max	Finds the largest value in the collection.
Min	Finds the smallest value in the collection.
Sum	Calculates sum of the values in the collection.

Aggregate:

The Aggregate method performs an accumulate operation. Aggregate extension method has the following overload methods:

Aggregate() method overloads:

```
public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,
                                         Func<TSource, TSource, TSource> func);

public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource>
source,
                                                         TAccumulate seed,
                                                         Func<TAccumulate, TSource, TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<TSource>
source,
                                                         TAccumulate seed,
                                                         Func<TAccumulate, TSource, TAccumulate> func,
                                                         Func<TAccumulate, TResult> resultSelector);
```

The following example demonstrates Aggregate method that returns comma separated elements of the string list.

Example: Aggregate operator in method syntax C#

```
IList<String> strList = new List<String>() { "One", "Two", "Three", "Four", "Five"};
var commaSeparatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);
Console.WriteLine(commaSeparatedString);
```

Output:



In the above example, Aggregate extension method returns comma separated strings from strList collection. The following image illustrates the whole aggregate operation performed in the above example.

Aggregate extension method

As per the above figure, first item of strList "One" will be pass as s1 and rest of the items will be passed as s2. The lambda expression `(s1, s2) => s1 + ", " + s2` will be treated like `s1 = s1 + ", " + s1` where s1 will be accumulated for each item in the collection. Thus, Aggregate method will return comma separated string.

Example: Aggregate operator in method syntax VB.Net

```
Dim strList As IList(Of String) = New List(Of String) From {
    "One",
    "Two",
    "Three",
    "Four",
```

```

        "Five"
    }

Dim commaSeparatedString = strList.Aggregate(Function(s1, s2) s1 + ", " + s2)

```

Aggregate method with seed value:

The second overload method of Aggregate requires first parameter for seed value to accumulate. Second parameter is Func type delegate:

```

TAccumulate Aggregate<TSource, TAccumulate>(TAccumulate seed,
Func<TAccumulate, TSource, TAccumulate> func);

```

The following example uses string as a seed value in the Aggregate extension method.

Example: Aggregate method with seed value in C#

```

// Student collection
IList<Student> studentList = new List<Student>>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};

string commaSeparatedStudentNames = studentList.Aggregate<Student, string>(
    "Student Names: ", // seed value
    (str, s) => str += s.StudentName + ", " );

Console.WriteLine(commaSeparatedStudentNames);

```

Example: Aggregate method with seed value in VB.Net

```

// Student collection
Dim studentList = New List(Of Student) From {
    New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},
    New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},
    New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},
    New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},
    New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}
}

Dim commaSeparatedStudentNames = studentList.Aggregate(Of String)(
    "Student Names: ",
    Function(str, s) str + s.StudentName + ", ")

Console.WriteLine(commaSeparatedStudentNames);

```

Output:

```
Student Names: John, Moin, Bill, Ram, Ron,
```

In the above example, the first parameter of the Aggregate method is the "Student Names: " string that will be accumulated with all student names. The comma in the lambda expression will be passed as a second parameter.

The following example use Aggregate operator to add the age of all the students.

Example: Aggregate method with seed value in C#

```
// Student collection
IList<Student> studentList = new List<Student>>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};

int SumOfStudentsAge = studentList.Aggregate<Student, int>(0,
    (totalAge, s) => totalAge += s.Age );
```

Aggregate method with result selector:

Now, let's see third overload method that required the third parameter of the Func delegate expression for result selector, so that you can formulate the result.

Consider the following example.

Example: Aggregate method C#

```
IList<Student> studentList = new List<Student>>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};

string commaSeparatedStudentNames = studentList.Aggregate<Student, string, string>(
    String.Empty, // seed value
    (str, s) => str += s.StudentName + ",", //
    returns result using seed value, String.Empty goes to lambda expression as str
    str => str.Substring(0, str.Length - 1 )); //
result selector that removes last comma

Console.WriteLine(commaSeparatedStudentNames);
```

Element Operators: First & FirstOrDefault

The First and FirstOrDefault method returns an element from the zeroth index in the collection i.e. the first element. Also, it returns an element that satisfies the specified condition.

Element Operators	Description
First	Returns the first element of a collection, or the first element that satisfies a condition.
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if index is out of range.

First and FirstOrDefault has two overload methods. The first overload method doesn't take any input parameter and returns the first element in the collection. The second overload method takes the lambda expression as predicate delegate to specify a condition and returns the first element that satisfies the specified condition.

Overload methods of First and FirstOrDefault - C#

```
public static TSource First<TSource>(this IEnumerable<TSource> source);

public static TSource First<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

The First() method returns the first element of a collection, or the first element that satisfies the specified condition using lambda expression or Func delegate. If a given collection is empty or does not include any element that satisfied the condition then it will throw InvalidOperationException exception.

The FirstOrDefault() method does the same thing as First() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

The following example demonstrates First() method.

Example: LINQ First() - C#

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };
IList<string> emptyList = new List<string>();

Console.WriteLine("1st Element in intList: {0}", intList.First());
Console.WriteLine("1st Even Element in intList: {0}", intList.First(i => i % 2 == 0));
```

```

Console.WriteLine("1st Element in strList: {0}", strList.First());

Console.WriteLine("emptyList.First() throws an InvalidOperationException");
Console.WriteLine("-----");
Console.WriteLine(emptyList.First());

```

Output:

```
1st Element in intList: 7
```

```
1st Even Element in intList: 10
```

```
1st Element in strList:
```

```
emptyList.First() throws an InvalidOperationException
```

```
-----
```

```
Run-time exception: Sequence contains no elements...
```

The following example demonstrates FirstOrDefault() method.

Example: LINQ FirstOrDefault() - C#

```

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };
IList<string> emptyList = new List<string>();

Console.WriteLine("1st Element in intList: {0}", intList.FirstOrDefault());
Console.WriteLine("1st Even Element in intList: {0}",
    intList.FirstOrDefault(i => i % 2 == 0));

Console.WriteLine("1st Element in strList: {0}", strList.FirstOrDefault());

Console.WriteLine("1st Element in emptyList: {0}", emptyList.FirstOrDefault());

```

Output:

```
1st Element in intList: 7
```

```
1st Even Element in intList: 10
```

```
1st Element in strList:
```

```
1st Element in emptyList:
```

Be careful while specifying condition in First() or FirstOrDefault(). First() will throw an exception if a collection does not include any element that satisfies the specified condition or includes null element.

If a collection includes null element then FirstOrDefault() throws an exception while evaluating the specified condition. The following example demonstrates this.

Example: LINQ First() & FirstOrDefault() - C#

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("1st Element which is greater than 250 in intList: {0}",
    intList.First( i > 250));

Console.WriteLine("1st Even Element in intList: {0}",
    strList.FirstOrDefault(s => s.Contains("T")));
```

Element Operators : Last & LastOrDefault

Element Operators	Description
Last	Returns the last element from a collection, or the last element that satisfies a condition
LastOrDefault	Returns the last element from a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.

Last and LastOrDefault has two overload methods. One overload method doesn't take any input parameter and returns last element from the collection. Second overload method takes a lambda expression to specify a condition and returns last element that satisfies the specified condition.

Overload methods of Last and LastOrDefault - C#

```
public static TSource Last<TSource>(this IEnumerable<TSource> source);
```

```
public static TSource Last<TSource>(this IEnumerable<TSource> source, Func<TSource, bool>
predicate);
```

```
public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source);
```

```
public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source,
Func<TSource, bool> predicate);
```

The Last() method returns the last element from a collection, or the last element that satisfies the specified condition using lambda expression or Func delegate. If a given collection is empty or does not include any element that satisfied the condition then it will throw InvalidOperationException exception.

The `LastOrDefault()` method does the same thing as `Last()` method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

The following example demonstrates `Last()` method.

Example: LINQ Last() - C#

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };
IList<string> emptyList = new List<string>();

Console.WriteLine("Last Element in intList: {0}", intList.Last());

Console.WriteLine("Last Even Element in intList: {0}", intList.Last(i => i % 2 == 0));

Console.WriteLine("Last Element in strList: {0}", strList.Last());

Console.WriteLine("emptyList.Last() throws an InvalidOperationException");
Console.WriteLine("-----");
Console.WriteLine(emptyList.Last());
```

Output:

Last Element in intList: 87

Last Even Element in intList: 50

Last Element in strList: Five

emptyList.Last() throws an InvalidOperationException

Run-time exception: Sequence contains no elements...

The following example demonstrates `LastOrDefault()` method.

Example: LINQ LastOrDefault() - C#

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };
IList<string> emptyList = new List<string>();

Console.WriteLine("Last Element in intList: {0}", intList.LastOrDefault());

Console.WriteLine("Last Even Element in intList: {0}",
    intList.LastOrDefault(i => i % 2 == 0));

Console.WriteLine("Last Element in strList: {0}", strList.LastOrDefault());
```



```
Console.WriteLine("Last Element in emptyList: {0}", emptyList.LastOrDefault());
```

Output:

```
Last Element in intList: 7
```

```
Last Even Element in intList: 10
```

```
Last Element in strList:
```

```
Last Element in emptyList:
```

Be careful while specifying condition in Last() or LastOrDefault(). Last() will throw an exception if a collection does not include any element that satisfies the specified condition or includes null element.

If a collection includes null element then LastOrDefault() throws an exception while evaluating the specified condition. The following example demonstrates this.

Example: LINQ Last() & LastOrDefault() - C#

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("Last Element which is greater than 250 in intList: {0}",
    intList.Last(i => i > 250));

Console.WriteLine("Last Even Element in intList: {0}",
    strList.LastOrDefault(s => s.Contains("T")));
```

Partitioning Operators: Skip & SkipWhile

Partitioning operators split the sequence (collection) into two parts and return one of the parts.

Method	Description
Skip	Skips elements up to a specified position starting from the first element in a sequence.
SkipWhile	Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence.
Take	Takes elements up to a specified position starting from the first element in a sequence.

Method	Description
TakeWhile	Returns elements from the first element until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition then returns an empty collection.

Skip:

The Skip() method skips the specified number of element starting from first element and returns rest of the elements.

Example: Skip() - C#

```
IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };
var newList = strList.Skip(2);
foreach(var str in newList)
    Console.WriteLine(str);
```

Output:



Skip operator in Query Syntax:

The Skip & SkipWhile operator is **Not Supported in C# query syntax**. However, you can use Skip/SkipWhile method on a query variable or wrap whole query into brackets and then call Skip/SkipWhile.

The following example demonstrates skip operator in query syntax - VB.NET

Example: Skip operator in VB.Net

```
Dim skipResult = From s In studentList
                  Skip 3
                  Select s
```

SkipWhile:

As the name suggests, the SkipWhile() extension method in LINQ skip elements in the collection till the specified condition is true. It returns a new collection that

includes all the remaining elements once the specified condition becomes false for any element.

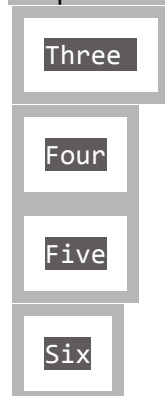
The `SkipWhile()` method has two overload methods. One method accepts the predicate of `Func<TSource, bool>` type and other overload method accepts the predicate `Func<TSource, int, bool>` type that pass the index of an element.

In the following example, `SkipWhile()` method skips all elements till it finds a string whose length is equal or more than 4 characters.

Example: SkipWhile in C#

```
IList<string> strList = new List<string>() {  
    "One",  
    "Two",  
    "Three",  
    "Four",  
    "Five",  
    "Six" };  
  
var resultList = strList.SkipWhile(s => s.Length < 4);  
  
foreach(string str in resultList)  
    Console.WriteLine(str);
```

Output:



In the above example, `SkipWhile()` skips first two elements because their length is less than 3 and finds third element whose length is equal or more than 4. Once it finds any element whose length is equal or more than 4 characters then it will not skip any other elements even if they are less than 4 characters.

Now, consider the following example where `SkipWhile()` does not skip any elements because the specified condition is false for the first element.

Example: SkipWhile in C#

```
IList<string> strList = new List<string>() {  
    "Three",  
    "One",  
    "Two",  
    "Four",
```

```

        "Five",
        "Six" };

var resultList = strList.SkipWhile(s => s.Length < 4);

foreach(string str in resultList)
    Console.WriteLine(str);

```

Output:

```

Three
One
Two
Four
Five
Six

```

The second overload of SkipWhile passes an index of each elements. Consider the following example.

Example: SkipWhile with index in C#

```

IList<string> strList = new List<string>() {
    "One",
    "Two",
    "Three",
    "Four",
    "Five",
    "Six" };

var result = strList.SkipWhile((s, i) => s.Length > i);

foreach(string str in result)
    Console.WriteLine(str);

```

Output:

```

Five
Six

```

In the above example, the lambda expression includes element and index of an elements as a parameter. It skips all the elements till the length of a string element is greater than it's index.

Skip/SkipWhile operator in Query Syntax:

Skip & SkipWhile operator is **NOT Supported in C# query syntax**. However, you can use Skip/SkipWhile method on a query variable or wrap whole query into brackets and then call Skip/SkipWhile().

Example: SkipWhile method in VB.Net

```
Dim strList = New List(Of string) From {  
    "One",  
    "Two",  
    "Three",  
    "Four",  
    "Five",  
    "Six" }  
  
Dim skipWhileResult = From s In studentList  
    Skip While s.Length < 4  
    Select s
```

Output:

Three

Four

Five

Six

Partitioning Operators: Take & TakeWhile

Partitioning operators split the sequence (collection) into two parts and returns one of the parts.

Method	Description
Skip	Skips elements up to a specified position starting from the first element in a sequence.
SkipWhile	Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence.
Take	Takes elements up to a specified position starting from the first element in a sequence.

Method	Description
TakeWhile	Returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

Take:

The Take() extension method returns the specified number of elements starting from the first element.

Example: Take() in C#

```
IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };
var newList = strList.Take(2);
foreach(var str in newList)
    Console.WriteLine(str);
```

Output:



Take & TakeWhile operator is **Not Supported in C# query syntax**. However, you can use Take/TakeWhile method on query variable or wrap whole query into brackets and then call Take/TakeWhile().

Example: Take operator in query syntax VB.Net

```
Dim takeResult = From s In studentList
                  Take 3
                  Select s
```

TakeWhile:

The TakeWhile() extension method returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

The TakeWhile method has two overload methods. One method accepts the predicate of `Func<TSource, bool>` type and the other overload method accepts the predicate `Func<TSource, int, bool>` type that passes the index of element.

In the following example, TakeWhile() method returns a new collection that includes all the elements till it finds a string whose length less than 4 characters.

Example: TakeWhile in C#

```
IList<string> strList = new List<string>() {  
    "Three",  
    "Four",  
    "Five",  
    "Hundred" };  
  
var result = strList.TakeWhile(s => s.Length > 4);  
  
foreach(string str in result)  
    Console.WriteLine(str);
```

Output:



Three

In the above example, TakeWhile() includes only first element because second string element does not satisfied the condition.

TakeWhile also passes an index of current element in predicate function. Following example of TakeWhile method takes elements till length of string element is greater than it's index

Example: TakeWhile in C#:

```
IList<string> strList = new List<string>() {  
    "One",  
    "Two",  
    "Three",  
    "Four",  
    "Five",  
    "Six" };  
  
var resultList = strList.TakeWhile((s, i) => s.Length > i);  
  
foreach(string str in resultList)  
    Console.WriteLine(str);
```

Output:



One



Two

Lab No 9:

Delegates

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate –

```
public delegate int MyDelegate (string s);
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is –

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

Action and Func Delegates in C#

The Func and Action generic delegates were introduced in the .NET Framework version 3.5.

Whenever we want to use delegates in our examples or applications, typically we use the following procedure:

- Define a custom delegate that matches the format of the method.
- Create an instance of a delegate and point it to a method.
- Invoke the method.

But, using these 2 Generics delegates we can simply eliminate the above procedure.

Since both the delegates are generic, you will need to specify the underlying types of each parameters as well while pointing it to a function. For example Action<type,type,type.....>

Action<>

- The Generic Action<> delegate is defined in the System namespace of microlib.dll
- This Action<> generic delegate, points to a method that takes up to 16 Parameters and returns void.

Func<>

- The generic Func<> delegate is used when we want to point to a method that returns a value.
- This delegate can point to a method that takes up to 16 Parameters and returns a value.
- Always remember that the final parameter of Func<> is always the return value of the method. (For example Func<int, int, string>, this version of the Func<> delegate will take 2 int parameters and returns a string value.)

Let's look at an example to see the use of both Delegates.

Create a new project named "FuncAndActionDelegates" and create a new class that holds all your methods.

MethodCollections.cs

```
1. class MethodCollections
2. {
3.
4.     //Methods that takes parameters but returns nothing:
5.
6.     public static void PrintText()
7.     {
8.         Console.WriteLine("Text Printed with the help of Action");
9.     }
10.    public static void PrintNumbers(int start, int target)
11.    {
12.        for (int i = start; i <= target; i++)
13.        {
14.            Console.Write(" {0}",i);
15.        }
16.        Console.WriteLine();
17.    }
18.    public static void Print(string message)
19.    {
20.        Console.WriteLine(message);
21.    }
22.
23.    //Methods that takes parameters and returns a value:
24.
25.    public static int Addition(int a, int b)
26.    {
27.        return a + b;
28.    }
29.
30.    public static string DisplayAddition(int a, int b)
31.    {
32.        return string.Format("Addition of {0} and {1} is {2}",a,b,a+b);
33.    }
34.
35.    public static string ShowCompleteName(string firstName, string lastName)
36.    {
37.        return string.Format("Your Name is {0} {1}",firstName,lastName);
38.    }
39.    public static int ShowNumber()
40.    {
```

```

41.         Random r = new Random();
42.         return r.Next();
43.     }
44. }

```

Program.cs

```

1. class Program
2. {
3.     static void Main(string[] args)
4.     {
5.         Action printText = new Action(MethodCollections.PrintText);
6.         Action<string> print = new Action<string>(MethodCollections.Print);
7.         Action<int, int> printNumber = new Action<int, int>(MethodCollections.PrintNumber);
8.
9.         Func<int, int, int> add1 = new Func<int, int, int>(MethodCollections.Addition);
10.        Func<int, int, string> add2 = new Func<int, int, string>(MethodCollections.DisplayAddition);
11.        Func<string, string, string> completeName = new Func<string, string, string>(MethodCollections.ShowCompleteName);
12.        Func<int> random = new Func<int>(MethodCollections.ShowNumber);
13.
14.        Console.WriteLine("\n***** Action<> Delegate Methods *****\n");
15.        printText(); //Parameter: 0 , Returns: nothing
16.        print("Abhishek"); //Parameter: 1 , Returns: nothing
17.        printNumber(5, 20); //Parameter: 2 , Returns: nothing
18.        Console.WriteLine();
19.        Console.WriteLine("***** Func<> Delegate Methods *****\n");
20.        int addition = add1(2, 5); //Parameter: 2 , Returns: int
21.        string addition2 = add2(5, 8); //Parameter: 2 , Returns: string
22.        string name = completeName("Saim", "Ali"); //Parameter: 2 , Returns: string
23.        int randomNumbers = random(); //Parameter: 0 , Returns: int
24.
25.        Console.WriteLine("Addition: {0}", addition);
26.        Console.WriteLine(addition2);
27.        Console.WriteLine(name);
28.        Console.WriteLine("Random Number is: {0}", randomNumbers);
29.
30.        Console.ReadLine();
31.    }
32. }

```

Lab No 10:

Multithreading in C#

Multithreading is a feature provided by the operating system that enables your application to have more than one execution path at the same time. Technically, multithreaded programming requires a multitasking operating system.

Let's understand this concept with a very basic example. Everyone has used Microsoft Word. It takes input from the user and displays it on the screen in one thread while it continues to check spelling and

grammatical mistakes in another thread and at the same time another thread saves the document automatically at regular intervals.

Now let's understand a program. In this program we will try to implement the concept of multithreading and will define multiple concurrent execution paths.

```
1  using System;
2  using System.Threading;
3
4  class MulThread
5  {
6      static void Main()
7      {
8          Thread firstthread= new Thread(new ThreadStart(func1));
9          Thread secondthread= new Thread(new ThreadStart(func2));
10         firstthread.Start();
11         secondthread.Start();
12     }
13     public static void func1()
14     {
15         for (int i=1;i<=5;i++)
16         {
17             Console.WriteLine("Function 1 writes:{0}",i);
18         }
19     }
20     public static void func2()
21     {
22         for (int i=10;i>5;i--)
23         {
24             Console.WriteLine("Function 2 writes:{0}",i);
25         }
26     }
27 }
```

Now through the use of threads, we cannot be sure of the sequence of the output program. Now if you write this program without the threading concept, like this:

```
6      static void Main()
7      {
8          func1();
9          func2();
10     }
```

You will see that func2() will execute only after func1() and in the first program you will be able to execute both functions simultaneously.

The thread class provides a number of useful methods and properties to control and manage thread execution.

Most Commonly used Static Member of System.Threading.Thread class

The following are the most commonly used static members of the System.Threading.Thread class:

- **CurrentThread():** gives a reference to the currently executing thread
- **Thread.Sleep():** cause the currently executing thread to pause temporarily for the specified amount of time

```

6      static void Main()
7      {
8          Console.WriteLine("Before calling Thread.Sleep() method");
9          Thread.Sleep(1000); // blocks the current executing thread for 1 second.
10         Console.WriteLine("After calling Thread.Sleep() method");
11     }

```

When you execute the preceding program, you will notice a delay of 1 second between the printing of the two lines.

Most Common Instance Member of the System.Threading.Thread class

The following are the most common instance members of the System.Threading.Thread class:

- **Name**
A property of string type used to get/set the friendly name of the thread instance.
- **Priority**
A property of type System.Threading.ThreadPriority to schedule the priority of threads.
- **IsAlive**
A Boolean property indicating whether the thread is alive or terminated.
- **ThreadState**
A property of type System.Threading.ThreadState, used to get the value containing the state of the thread.
- **Start()**
Starts the execution of the thread.
- **Abort()**
Allows the current thread to stop the execution of the thread permanently.
- **Suspend()**
Pauses the execution of the thread temporarily.
- **Resume()**
Resumes the execution of a suspended thread.
- **Join()**
Make the current thread wait for another thread to finish.

Lab No 11:

Threads and Thread Synchronization in C#

Introduction

Although C# contains many innovative features, one of the most exciting is its built in support for multithreaded programming.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

*Multithreaded applications provide the illusion that numerous activities are happening at more or less the same time. But the reality is the CPU uses something known as "**Time Slice**" to switch between different threads.*

The principal advantage of multithreading is that it enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs. But using too many threads in our programs can actually "degrade" performance, as the CPU must switch between the active threads in the process which takes time. When a thread's time slice is up, the existing thread is suspended to allow other thread to perform its business. For a thread to remember what was happening before it was kicked out of the way, it writes information to its local storage and it is also provided with a separate call stack, which again put extra load on the CPU.

- from **Herbert Schildt's** book "**C# 2.0 The complete Reference**". chapter 22

Multithreading Fundamentals

There are two distinct types of multitasking: **process-based** and **thread-based**.

The differences between process-based and thread-based multitasking can be summarized like this:

Process-based multitasking handles the concurrent execution of programs, while Thread-based multitasking deals with the concurrent execution of pieces of the same program.

Process-based: Example — running word processor at the same time you are browsing the net.

Thread-based: Example — A text editor can be formatting text at the same time that it is printing.

Simply we can define a thread as a line of execution within a process and it can exist in any of these several states.

It can be **running**. It can be **ready to run** as soon as it gets CPU time. A running thread can be **suspended**, which is a temporary halt to its execution. It can later be **resumed**. A thread can be **blocked** when waiting for a resource. A thread can be **terminated**, in which case its execution ends and cannot be resumed.

The .NET Framework defines two types of threads: **foreground** and **background**.

By default when you create a thread, it is a foreground thread, but you can change it to a background thread. The only difference between a foreground and background thread is that a background thread will be automatically terminated when all foreground threads in its process have stopped.

The "**Foreground**" threads have the ability to prevent the current application from terminating. The CLR will not shutdown an application until all foreground threads have ended. The "**Background**" threads are viewed by the CLR as expandable paths of execution that can be ignored at any point of time even if they are laboring over some unit of work. Thus, if all foreground threads have terminated, any background threads operating are automatically killed when the application terminates.

All processes have at least one thread of execution, which is usually called the **main thread** because it is the one that is executed when your program begins. From the main thread you can create other threads. The classes that support multithreaded programming are defined in the **System.Threading** namespace. Thus, you will usually include this statement at the start of any multithreaded program:

```
using System.Threading;
```

Creating a Thread

To create a thread, you instantiate an object of type **Thread**. **Thread** defines the following constructor:

```
public Thread( ThreadStart entrypoint)
```

Here, **entrypoint** is the name of the method that will be called to begin execution of the thread. **ThreadStart** is a delegate defined by the .NET Framework as shown here:

```
public delegate void ThreadStart()
```

Thus, your entrypoint method must have a **void** return type and take no arguments. Once created, the new thread will not start running until you call its **Start()** method, which is defined by **Thread**. The **Start()** method is shown here:

```
public void Start()
```

Once started, the thread will run until the method specified by **entryPoint** returns. Thus, when **entryPoint** returns, the thread automatically stops. If you try to call **Start()** on a thread that has already been started, a **ThreadStateException** will be thrown.

Example

Hide Shrink ▲ Copy Code

```
using System;
using System.Threading;
namespace CSharpThreadExample
{
    class Program
    {
        public static void run()
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("In thread " + Thread.CurrentThread.Name + i);
                Thread.Sleep(1000);
            }
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread Starting");
            Thread.CurrentThread.Name = "Main ";

            Thread t1 = new Thread(new ThreadStart(run));
            t1.Name = "Child";
            t1.Start();

            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("In thread " + Thread.CurrentThread.Name + i);
```

```

        Thread.Sleep(1000);
    }
    Console.WriteLine("Main Thread Terminates");
    Console.Read();
}
}
}

```

Notice the call to **Sleep()**, which is a **static** method defined by **Thread**.

The **Sleep()** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. The form used by the program is shown here:

```
public static void Sleep(int milliseconds)
```

The number of milliseconds to suspend is specified in milliseconds. If milliseconds is zero, the calling thread is suspended only to allow a waiting thread to execute.

Here's the output:



The screenshot shows a console window with the following output:

```

Main Thread Starting
In thread Main 0
In thread Child 0
In thread Child 1
In thread Main 1
In thread Child 2
In thread Main 2
In thread Main 3
In thread Child 3
In thread Child 4
In thread Main 4
Main Thread Terminates

```

Each thread maintains a private set of structures that the O/S uses to save information (the thread context) when the thread is not running including the value of CPU registers. It also maintains the priority levels. We can assign higher priority to a thread of more important task than to a thread which works in background. In all cases time slice allocated to each thread is relatively short, so that the end user has the perception that all the threads (and all the applications) are running concurrently.

O/S has thread scheduler which schedules existing threads and preempts the running thread when its time slice expires. We make the most use of multithreading when we allocate distinct threads to tasks that have different priorities or that take a lot of time to complete.

The main problem with threads is that they compete for shared resource, a resource can be a variable, a database connection, a H/W device. We must synchronize the access to such resources — otherwise we will result in deadlock situations. A thread terminates when the task provided to it terminates or when the thread is programmatically killed by calling **Abort()**. An application as a whole terminates only when all its threads terminate.

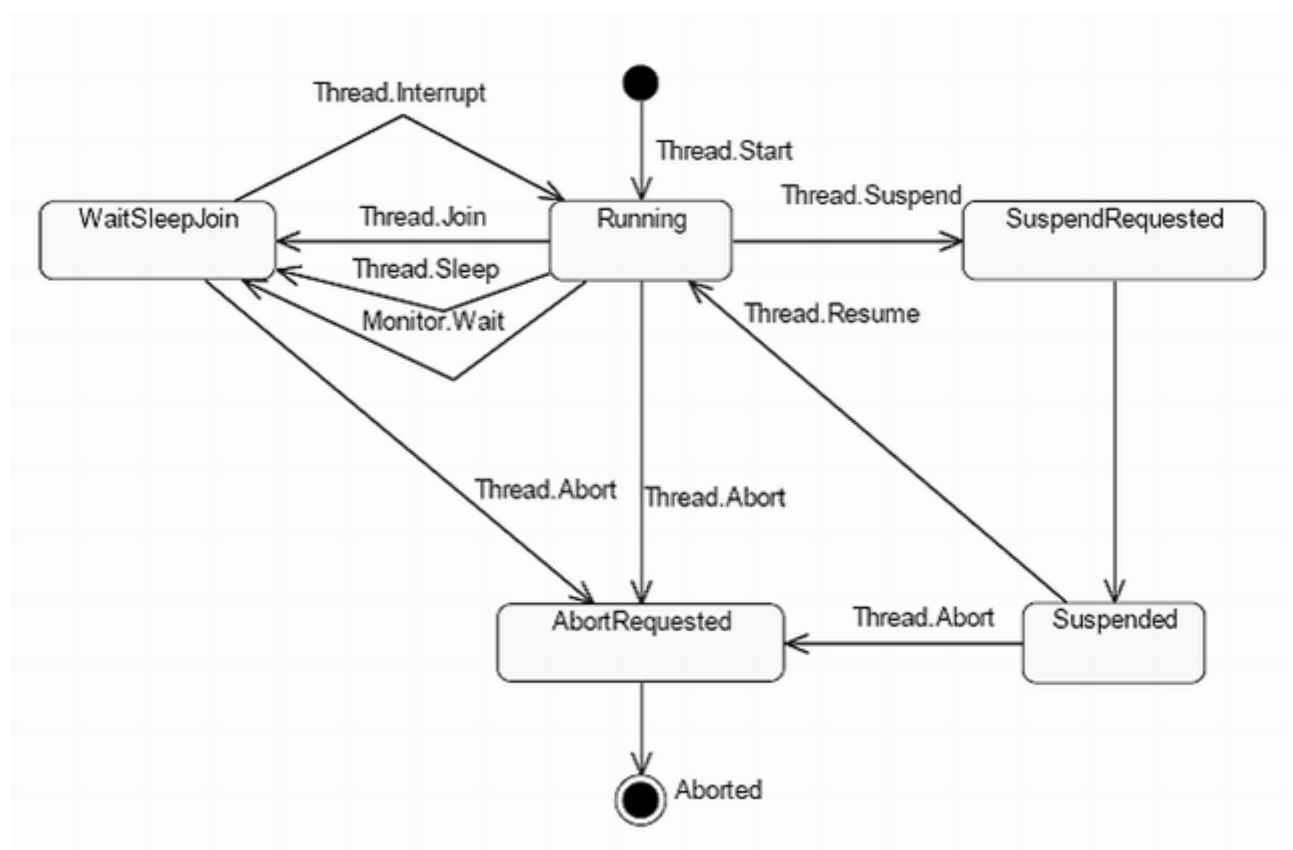
Methods

- **Suspend()** -> Suspends the execution of a thread till **Resume()** is called on that.
- **Resume()** -> Resumes a suspended thread. Can throw exceptions for bad state of the thread.
- **Sleep()** -> A thread can suspend itself by calling **Sleep()**. Takes parameter in form of milliseconds. We can use a special timeout 0 to terminate the current time slice and give other thread a chance to use CPU time
- **Join()** -> Called on a thread makes other threads wait for it till it finishes its task.

States of a Thread

States of a thread can be checked using `ThreadState` enumerated property of the `Thread` object which contains a different value for different states.

- Aborted -> Aborted already.
- AbortRequested -> Responding to an `Abort()` request.
- Background -> Running in background. Same as `IsBackground` property.
- Running -> Running after another thread has called the `start()`
- Stopped -> After finishing `run()` or `Abort()` stopped it.
- Suspended -> Suspended after `Suspend()` is called.
- Unstarted -> Created but `start()` has not been called.
- WaitSleepJoin -> `Sleep()/Wait()` on itself and `join()` on another thread. If a thread Thread1 calls `sleep()` on itself and calls `join()` on the thread Thread2 then it enters WaitSleepJoin state. The thread exists in this state till the timeout expires or another thread invokes `Interrupt()` on it. It is wise to check the state of a thread before calling methods on it to avoid `ThreadStateException`.



This picture describes in detail about the states of the thread [Collected from "**Thinking in C#**" by **Bruce Eckel**]

Properties of a Thread

- `Thread.CurrentThread` -> Static method gives the reference of the thread object which is executing the current code.
- Name -> Read/Write Property used to get and set the name of a thread
- ThreadState -> Property used to check the state of a thread.
- Priority -> Property used to check for the priority level of a thread.
- IsAlive -> Returns a Boolean value stating whether the thread is alive or not.
- IsBackground -> Returns a Boolean value stating the running in background or foreground.

PriorityLevels of Thread

Priority levels of thread is set or checked by using an enumeration i.e. **ThreadPriority**. The valid values are for this enumeration are;

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

Synchronization in Threads

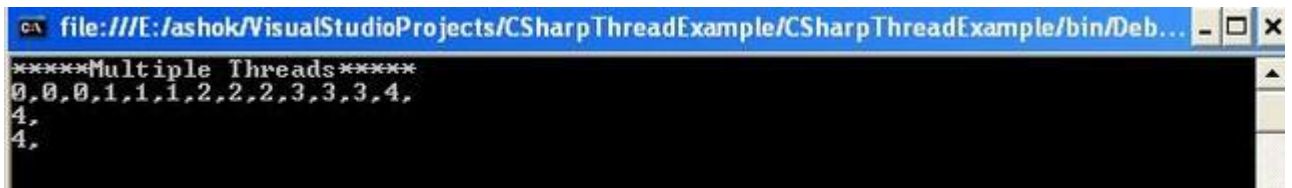
When we have multiple threads that share data, we need to provide synchronized access to the data. We have to deal with synchronization issues related to concurrent access to variables and objects accessible by multiple threads at the same time. This is controlled by giving one thread a chance to acquire a lock on the shared resource at a time. We can think it like a box where the object is available and only one thread can enter into and the other thread is waiting outside the box until the previous one comes out.

Hide Shrink ▲ Copy Code

```
using System;
using System.Threading;
namespace CSharpThreadExample
{
    class Program
    {
        static void Main(string[] arg)
        {
            Console.WriteLine("*****Multiple Threads*****");
            Printer p=new Printer();
            Thread[] Threads=new Thread[3];
            for(int i=0;i<3;i++)
            {
                Threads[i]=new Thread(new ThreadStart(p.PrintNumbers));
                Threads[i].Name="Child "+i;
            }
            foreach(Thread t in Threads)
                t.Start();

            Console.ReadLine();
        }
    }
    class Printer
    {
        public void PrintNumbers()
        {
            for (int i = 0; i < 5; i++)
            {
                Thread.Sleep(100);
                Console.Write(i + ",");
            }
            Console.WriteLine();
        }
    }
}
```

In the above example, we have created three threads in the main method and all the threads are trying to use the **PrintNumbers()** method of the same **Printer** object to print to the console. Here we get this type of output:



Now we can see, as the thread scheduler is swapping threads in the background each thread is telling the **Printer** to print the numerical data. We are getting inconsistent output as the access of these threads to the **Printer** object is synchronized. There are various synchronization options which we can use in our programs to enable synchronization of the shared resource among multiple threads.

Using the Lock Keyword

In C# we use `lock(object)` to synchronize the shared object.

Syntax:

```
lock (objecttobelocked)    {  
    objecttobelocked.somemethod();  
}
```

Here `objecttobelocked` is the object reference which is used by more than one thread to call the method on that object. The `lock` keyword requires us to specify a token (an object reference) that must be acquired by a thread to enter within the lock scope. When we are attempting to lock down an instance level method, we can simply pass the reference to that instance. (We can use `this` keyword to lock the current object) Once the thread enters into a lock scope, the lock token (object reference) is inaccessible by other threads until the lock is released or the lock scope has exited.

If we want to lock down the code in a static method, we need to provide the `System.Type` of the respective class.

Converting the Code to Enable Synchronization using the Lock Keyword

```
public void PrintNumbers()  
{  
    lock (this)  
    {  
        for (int i = 0; i < 5; i++)  
        {  
            Thread.Sleep(100);  
            Console.Write(i + ",");  
        }  
        Console.WriteLine();  
    }  
}
```

OUTPUT



Using the Monitor Type

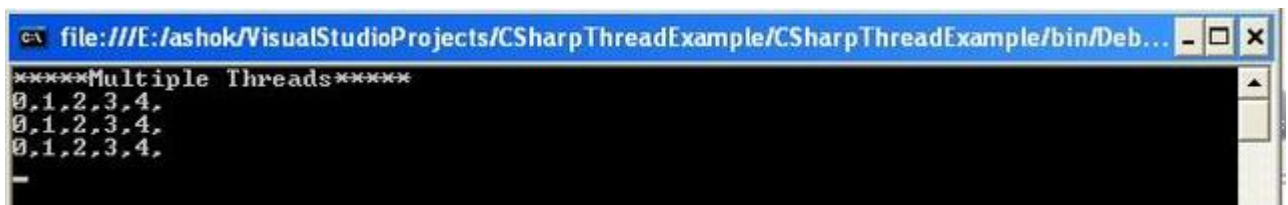
The C# lock keyword is just a notation for using `System.Threading.Monitor` class type. The `lock` scope actually resolves to the `Monitor` class after being processed by the C# compiler.

Converting the Code to Enable Synchronization using the Monitor Class

```
public void PrintNumbers()
{
    Monitor.Enter(this);
    try
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(100);
            Console.Write(i + ",");
        }
        Console.WriteLine();
    }
    finally
    {
        Monitor.Exit(this);
    }
}
```

`Monitor.Enter()` method is the ultimate recipient of the thread token. We need to write all code of the lock scope inside a try block. The finally clause ensures that the thread token is released(using the `Monitor.Exit()` method), regardless of any runtime exception.

OUTPUT



In this article I have tried to make the explanation as simple as possible. I have referred the book written by **Andrew Troelson** published by **Apress**. Please provide your valuable comments and suggestions for improvement of this article. I am always open for future upgradation of the article.

Lab No 12:

Socket Programming:

TcpClient Class

Provides client connections for TCP network services.

Namespace: [System.Net.Sockets](#)

Assembly: System (in System.dll)

Inheritance Hierarchy

[System.Object](#)





System.Net.Sockets.TcpClient

Syntax






C#







```
public class TcpClient : IDisposable
```

Constructors









	Name	Description
	TcpClient()	Initializes a new instance of the TcpClient class.
	TcpClient(AddressFamily)	Initializes a new instance of the TcpClient class with the specified family.
	TcpClient(IPEndPoint)	Initializes a new instance of the TcpClient class and binds it to the specified local endpoint.
	TcpClient(String, Int32)	Initializes a new instance of the TcpClient class and connects to the specified port on the specified host.














Properties

	Name	Description
	Active	Gets or set a value that indicates whether a connection has been made.
	Available	Gets the amount of data that has been received from the network and is available to be read.
	Client	Gets or sets the underlying Socket .
	Connected	Gets a value indicating whether the underlying Socket for a TcpClient is connected to a remote host.
	ExclusiveAddressUse	Gets or sets a Boolean value that specifies whether the TcpClient allows only one client to use a port.

	<u>LingerState</u>	Gets or sets information about the linger state of the associated socket.
	<u>NoDelay</u>	Gets or sets a value that disables a delay when send or receive buffers are not full.
	<u>ReceiveBufferSize</u>	Gets or sets the size of the receive buffer.
	<u>ReceiveTimeout</u>	Gets or sets the amount of time a TcpClient will wait to receive data once a read operation is initiated.
	<u>SendBufferSize</u>	Gets or sets the size of the send buffer.
	<u>SendTimeout</u>	Gets or sets the amount of time a TcpClient will wait for a send operation to complete successfully.

Methods

	Name	Description
	<u>BeginConnect(IPAddress, Int32, AsyncCallback, Object)</u>	Begins an asynchronous request for a remote host connection. The remote host is specified by an <u>IPAddress</u> and a port number (<u>Int32</u>).
	<u>BeginConnect(IPAddress[], Int32, AsyncCallback, Object)</u>	Begins an asynchronous request for a remote host connection. The remote host is specified by an <u>IPAddress</u> array and a port number (<u>Int32</u>).
	<u>BeginConnect(String, Int32, AsyncCallback, Object)</u>	Begins an asynchronous request for a remote host connection. The remote host is specified by a host name (<u>String</u>) and a port number (<u>Int32</u>).
	<u>Close()</u>	Disposes this TcpClient instance and requests that the underlying TCP connection be closed.
	<u>Connect(IPAddress, Int32)</u>	Connects the client to a remote TCP host using the specified IP address and port number.
	<u>Connect(IPAddress[], Int32)</u>	Connects the client to a remote TCP host using the specified IP addresses and port number.
	<u>Connect(IPEndPoint)</u>	Connects the client to a remote TCP host using the specified remote network endpoint.
	<u>Connect(String, Int32)</u>	Connects the client to the specified port on the specified host.

	ConnectAsync(IPAddress, Int32)	Connects the client to a remote TCP host using the specified IP address and port number as an asynchronous operation.
	ConnectAsync(IPAddress[], Int32)	Connects the client to a remote TCP host using the specified IP addresses and port number as an asynchronous operation.
	ConnectAsync(String, Int32)	Connects the client to the specified TCP port on the specified host as an asynchronous operation.
	Dispose()	Releases the managed and unmanaged resources used by the TcpClient.
	Dispose(Boolean)	Releases the unmanaged resources used by the TcpClient and optionally releases the managed resources.
	EndConnect(IAsyncResult)	Ends a pending asynchronous connection attempt.
	Equals(Object)	Determines whether the specified object is equal to the current object.(Inherited from Object .)
	Finalize()	Frees resources used by the TcpClient class.(Overrides Object.Finalize() .)
	GetHashCode()	Serves as the default hash function. (Inherited from Object .)
	GetStream()	Returns the NetworkStream used to send and receive data.
	GetType()	Gets the Type of the current instance.(Inherited from Object .)
	MemberwiseClone()	Creates a shallow copy of the current Object .(Inherited from Object .)
	ToString()	Returns a string that represents the current object.(Inherited from Object .)

Remarks

The TcpClient class provides simple methods for connecting, sending, and receiving stream data over a network in synchronous blocking mode.

In order for TcpClient to connect and exchange data, a [TcpListener](#) or [Socket](#) created with the TCP [ProtocolType](#) must be listening for incoming connection requests. You can connect to this listener in one of the following two ways:

- Create a TcpClient and call one of the three available [Connect](#) methods.
- Create a TcpClient using the host name and port number of the remote host. This constructor will automatically attempt a connection.

Notes to Inheritors:

To send and receive data, use the [GetStream](#) method to obtain a [NetworkStream](#). Call the [Write](#) and [Read](#) methods of the [NetworkStream](#) to send and receive data with the remote host. Use the [Close](#) method to release all resources associated with the TcpClient.

Examples

The following code example establishes a TcpClient connection.

```
C#
static void Connect(String server, String message)
{
    try
    {
        // Create a TcpClient.
        // Note, for this client to work you need to have a TcpServer
        // connected to the same address as specified by the server, port
        // combination.
        Int32 port = 13000;
        TcpClient client = new TcpClient(server, port);

        // Translate the passed message into ASCII and store it as a Byte array.
        Byte[] data = System.Text.Encoding.ASCII.GetBytes(message);

        // Get a client stream for reading and writing.
        // Stream stream = client.GetStream();

        NetworkStream stream = client.GetStream();

        // Send the message to the connected TcpServer.
        stream.Write(data, 0, data.Length);

        Console.WriteLine("Sent: {0}", message);

        // Receive the TcpServer.response.

        // Buffer to store the response bytes.
        data = new Byte[256];

        // String to store the response ASCII representation.
        String responseData = String.Empty;

        // Read the first batch of the TcpServer response bytes.
        Int32 bytes = stream.Read(data, 0, data.Length);
        responseData = System.Text.Encoding.ASCII.GetString(data, 0, bytes);
        Console.WriteLine("Received: {0}", responseData);

        // Close everything.
        stream.Close();
        client.Close();
    }
    catch (ArgumentNullException e)
    {
        Console.WriteLine("ArgumentNullException: {0}", e);
    }
    catch (SocketException e)
    {
        Console.WriteLine("SocketException: {0}", e);
    }

    Console.WriteLine("\n Press Enter to continue...");
    Console.Read();
}
```

Security

[SocketPermission](#)

Permission to establish an outgoing connection or accept an incoming request.

Any public static (**Shared** in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

TcpListener Class

Listens for connections from TCP network clients.

Namespace: [System.Net.Sockets](#)




Assembly: System (in System.dll)

Syntax




C#

```
public class TcpListener
```

Constructors














	Name	Description
	TcpListener(Int32)	Obsolete. Initializes a new instance of the TcpListener class that listens on the specified port.
	TcpListener(IPAddress, Int32)	Initializes a new instance of the TcpListener class that listens for incoming connection attempts on the specified local IP address and port number.
	TcpListener(IPEndPoint)	Initializes a new instance of the TcpListener class with the specified local endpoint.








Properties

	Name	Description
	Active	Gets a value that indicates whether TcpListener is actively listening for client connections.
	ExclusiveAddressUse	Gets or sets a Boolean value that specifies whether the TcpListener allows only one underlying socket to listen to a specific port.
	LocalEndPoint	Gets the underlying EndPoint of the current TcpListener.

	Server	Gets the underlying network Socket .
-----------------------------------------------------------------------------------	------------------------	------------------------------------------------------

Methods

	Name	Description
	AcceptSocket()	Accepts a pending connection request.
	AcceptSocketAsync()	Accepts a pending connection request as an asynchronous operation.
	AcceptTcpClient()	Accepts a pending connection request.
	AcceptTcpClientAsync()	Accepts a pending connection request as an asynchronous operation.
	AllowNatTraversal(Boolean)	Enables or disables Network Address Translation (NAT) traversal on a TcpListenerinstance.
	BeginAcceptSocket(AsyncCallback, Object)	Begins an asynchronous operation to accept an incoming connection attempt.
	BeginAcceptTcpClient(AsyncCallback, Object)	Begins an asynchronous operation to accept an incoming connection attempt.
	Create(Int32)	Creates a new TcpListener instance to listen on the specified port.
	EndAcceptSocket(IAsyncResult)	Asynchronously accepts an incoming connection attempt and creates a new Socket to handle remote host communication.
	EndAcceptTcpClient(IAsyncResult)	Asynchronously accepts an incoming connection attempt and creates a new TcpClient to handle remote host communication.
	Equals(Object)	Determines whether the specified object is equal to the current object.(Inherited from Object .)
	Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from Object .)
	GetHashCode()	Serves as the default hash function. (Inherited

		from Object .)
	GetType()	Gets the Type of the current instance.(Inherited from Object .)
	MemberwiseClone()	Creates a shallow copy of the current Object .(Inherited from Object .)
	Pending()	Determines if there are pending connection requests.
	Start()	Starts listening for incoming connection requests.
	Start(Int32)	Starts listening for incoming connection requests with a maximum number of pending connection.
	Stop()	Closes the listener.
	ToString()	Returns a string that represents the current object.(Inherited from Object .)

Remarks

The TcpListener class provides simple methods that listen for and accept incoming connection requests in blocking synchronous mode. You can use either a [TcpClient](#) or a [Socket](#) to connect with a TcpListener. Create a TcpListener using an [IPEndPoint](#), a Local IP address and port number, or just a port number. Specify [Any](#) for the local IP address and 0 for the local port number if you want the underlying service provider to assign those values for you. If you choose to do this, you can use the [LocalEndPoint](#) property to identify the assigned information, after the socket has connected.

Use the [Start](#) method to begin listening for incoming connection requests. [Start](#) will queue incoming connections until you either call the [Stop](#) method or it has queued [MaxConnections](#). Use either [AcceptSocket](#) or [AcceptTcpClient](#) to pull a connection from the incoming connection request queue. These two methods will block. If you want to avoid blocking, you can use the [Pending](#) method first to determine if connection requests are available in the queue.

Call the [Stop](#) method to close the TcpListener.

Examples

The following code example creates a TcpListener.

```
C#
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class MyTcpListener
{
    public static void Main()
    {
        TcpListener server=null;
        try
        {
            // Set the TcpListener on port 13000.
            Int32 port = 13000;
```

```

IPAddress localAddr = IPAddress.Parse("127.0.0.1");

// TcpListener server = new TcpListener(port);
server = new TcpListener(localAddr, port);

// Start listening for client requests.
server.Start();

// Buffer for reading data
Byte[] bytes = new Byte[256];
String data = null;

// Enter the listening loop.
while(true)
{
    Console.Write("Waiting for a connection... ");

    // Perform a blocking call to accept requests.
    // You could also use server.AcceptSocket() here.
    TcpClient client = server.AcceptTcpClient();
    Console.WriteLine("Connected!");

    data = null;

    // Get a stream object for reading and writing
    NetworkStream stream = client.GetStream();

    int i;

    // Loop to receive all the data sent by the client.
    while((i = stream.Read(bytes, 0, bytes.Length)) != 0)
    {
        // Translate data bytes to a ASCII string.
        data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
        Console.WriteLine("Received: {0}", data);

        // Process the data sent by the client.
        data = data.ToUpper();

        byte[] msg = System.Text.Encoding.ASCII.GetBytes(data);

        // Send back a response.
        stream.Write(msg, 0, msg.Length);
        Console.WriteLine("Sent: {0}", data);
    }

    // Shutdown and end connection
    client.Close();
}
}
catch(SocketException e)
{
    Console.WriteLine("SocketException: {0}", e);
}
finally
{
    // Stop listening for new clients.
    server.Stop();
}

Console.WriteLine("\nHit enter to continue...");

```

```

        Console.Read();
    }
}

```

UdpClient Class

Provides User Datagram Protocol (UDP) network services.

Namespace: [System.Net.Sockets](#)







Assembly: System (in System.dll)

Syntax


C#








```
public class UdpClient : IDisposable
```

Constructors








	Name	Description
	UdpClient()	Initializes a new instance of the UdpClient class.
	UdpClient(AddressFamily)	Initializes a new instance of the UdpClient class.
	UdpClient(Int32)	Initializes a new instance of the UdpClient class and binds it to the local port number provided.
	UdpClient(Int32, AddressFamily)	Initializes a new instance of the UdpClient class and binds it to the local port number provided.
	UdpClient(IPEndPoint)	Initializes a new instance of the UdpClient class and binds it to the specified local endpoint.
	UdpClient(String, Int32)	Initializes a new instance of the UdpClient class and establishes a default remote host.


















Properties










	Name	Description
	Active	Gets or sets a value indicating whether a default remote host has been established.

	Available	Gets the amount of data received from the network that is available to read.
	Client	Gets or sets the underlying network Socket .
	DontFragment	Gets or sets a Boolean value that specifies whether the UdpClient allows Internet Protocol (IP) datagrams to be fragmented.
	EnableBroadcast	Gets or sets a Boolean value that specifies whether the UdpClient may send or receive broadcast packets.
	ExclusiveAddressUse	Gets or sets a Boolean value that specifies whether the UdpClient allows only one client to use a port.
	MulticastLoopback	Gets or sets a Boolean value that specifies whether outgoing multicast packets are delivered to the sending application.
	Ttl	Gets or sets a value that specifies the Time to Live (TTL) value of Internet Protocol (IP) packets sent by the UdpClient.

Methods

	Name	Description
	AllowNatTraversal(Boolean)	Enables or disables Network Address Translation (NAT) traversal on a UdpClient instance.
	BeginReceive(AsyncCallback, Object)	Receives a datagram from a remote host asynchronously.
	BeginSend(Byte[], Int32, AsyncCallback, Object)	Sends a datagram to a remote host asynchronously. The destination was specified previously by a call to Connect .
	BeginSend(Byte[], Int32, IPEndPoint, AsyncCallback, Object)	Sends a datagram to a destination asynchronously. The destination is specified by a EndPoint .
	BeginSend(Byte[], Int32, String, Int32, AsyncCallback, Object)	Sends a datagram to a destination asynchronously. The destination is specified by the host name and port number.
	Close()	Closes the UDP connection.
	Connect(IPAddress, Int32)	Establishes a default remote host using the specified IP address and port number.

	<u>Connect(IPEndPoint)</u>	Establishes a default remote host using the specified network endpoint.
	<u>Connect(String, Int32)</u>	Establishes a default remote host using the specified host name and port number.
	<u>Dispose()</u>	Releases the managed and unmanaged resources used by the <code>UdpClient</code> .
	<u>Dispose(Boolean)</u>	Releases the unmanaged resources used by the <code>UdpClient</code> and optionally releases the managed resources.
	<u>DropMulticastGroup(IPAddress)</u>	Leaves a multicast group.
	<u>DropMulticastGroup(IPAddress, Int32)</u>	Leaves a multicast group.
	<u>EndReceive(IAsyncResult, IPEndPoint)</u>	Ends a pending asynchronous receive.
	<u>EndSend(IAsyncResult)</u>	Ends a pending asynchronous send.
	<u>Equals(Object)</u>	Determines whether the specified object is equal to the current object.(Inherited from <u>Object</u> .)
	<u>Finalize()</u>	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from <u>Object</u> .)
	<u>GetHashCode()</u>	Serves as the default hash function. (Inherited from <u>Object</u> .)
	<u>GetType()</u>	Gets the <u>Type</u> of the current instance.(Inherited from <u>Object</u> .)
	<u>JoinMulticastGroup(Int32, IPAddress)</u>	Adds a <code>UdpClient</code> to a multicast group.
	<u>JoinMulticastGroup(IPAddress)</u>	Adds a <code>UdpClient</code> to a multicast group.
	<u>JoinMulticastGroup(IPAddress, Int32)</u>	Adds a <code>UdpClient</code> to a multicast group with the specified Time to Live (TTL).
	<u>JoinMulticastGroup(IPAddress, IPAddress)</u>	Adds a <code>UdpClient</code> to a multicast group.
	<u>MemberwiseClone()</u>	Creates a shallow copy of the current <u>Object</u> .(Inherited from <u>Object</u> .)

	Receive(IPEndPoint)	Returns a UDP datagram that was sent by a remote host.
	ReceiveAsync()	Returns a UDP datagram asynchronously that was sent by a remote host.
	Send(Byte[], Int32)	Sends a UDP datagram to a remote host.
	Send(Byte[], Int32, IPEndPoint)	Sends a UDP datagram to the host at the specified remote endpoint.
	Send(Byte[], Int32, String, Int32)	Sends a UDP datagram to a specified port on a specified remote host.
	SendAsync(Byte[], Int32)	Sends a UDP datagram asynchronously to a remote host.
	SendAsync(Byte[], Int32, IPEndPoint)	Sends a UDP datagram asynchronously to a remote host.
	SendAsync(Byte[], Int32, String, Int32)	Sends a UDP datagram asynchronously to a remote host.
	ToString()	Returns a string that represents the current object.(Inherited from Object .)

Remarks

The `UdpClient` class provides simple methods for sending and receiving connectionless UDP datagrams in blocking synchronous mode. Because UDP is a connectionless transport protocol, you do not need to establish a remote host connection prior to sending and receiving data. You do, however, have the option of establishing a default remote host in one of the following two ways:

- Create an instance of the `UdpClient` class using the remote host name and port number as parameters.
- Create an instance of the `UdpClient` class and then call the [Connect](#) method.

You can use any of the send methods provided in the `UdpClient` to send data to a remote device. Use the [Receive](#) method to receive data from remote hosts.

`UdpClient` methods also allow you to send and receive multicast datagrams. Use the [JoinMulticastGroup](#) method to subscribe a `UdpClient` to a multicast group. Use the [DropMulticastGroup](#) method to unsubscribe a `UdpClient` from a multicast group.

Examples

The following example establishes a `UdpClient` connection using the host name `www.contoso.com` on port 11000. A small string message is sent to two separate remote host machines. The [Receive](#) method blocks execution until a message is received. Using the [IPEndPoint](#) passed to [Receive](#), the identity of the responding host is revealed.

```
C#
// This constructor arbitrarily assigns the local port number.
UdpClient udpClient = new UdpClient(11000);
try{
    udpClient.Connect("www.contoso.com", 11000);

    // Sends a message to the host to which you have connected.
```

```

Byte[] sendBytes = Encoding.ASCII.GetBytes("Is anybody there?");

udpClient.Send(sendBytes, sendBytes.Length);

// Sends a message to a different host using optional hostname and port
parameters.
UdpClient udpClientB = new UdpClient();
udpClientB.Send(sendBytes, sendBytes.Length, "AlternateHostMachineName",
11000);

//IPEndPoint object will allow us to read datagrams sent from any source.
IPEndPoint RemoteIpEndPoint = new IPEndPoint(IPAddress.Any, 0);

// Blocks until a message returns on this socket from a remote host.
Byte[] receiveBytes = udpClient.Receive(ref RemoteIpEndPoint);
string returnData = Encoding.ASCII.GetString(receiveBytes);

// Uses the IPEndPoint object to determine which of these two hosts
responded.
Console.WriteLine("This is the message you received " +
returnData.ToString());
Console.WriteLine("This message was sent from " +
RemoteIpEndPoint.Address.ToString() +
" on their port number " +
RemoteIpEndPoint.Port.ToString());

udpClient.Close();
udpClientB.Close();

}
catch (Exception e) {
Console.WriteLine(e.ToString());
}

```

Security

[SocketPermission](#)

To establish an outgoing connection or accept an incoming request.

Lab No 13:

Creational Patterns

Singleton

Definition

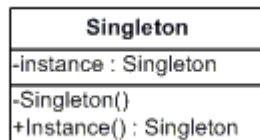
Ensure a class has only one instance and provide a global point of access to it.

Frequency of use:



Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Singleton (LoadBalancer)**
 - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
 - responsible for creating and maintaining its own unique instance.

Structural code in C#

This **structural** code demonstrates the Singleton pattern which assures only a single instance (the singleton) of the class can be created.

1.
2.

```

3. using System;
4.
5. namespace DoFactory.GangOfFour.Singleton.Structural
6. {
7.     /// <summary>
8.     /// MainApp startup class for Structural
9.     /// Singleton Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        static void Main()
17.        {
18.            // Constructor is protected -- cannot use new
19.            Singleton s1 = Singleton.Instance();
20.            Singleton s2 = Singleton.Instance();
21.
22.            // Test for same instance
23.            if (s1 == s2)
24.            {
25.                Console.WriteLine("Objects are the same instance");
26.            }
27.
28.            // Wait for user
29.            Console.ReadKey();
30.        }
31.    }
32.
33.    /// <summary>
34.    /// The 'Singleton' class
35.    /// </summary>
36.    class Singleton
37.    {
38.        private static Singleton _instance;
39.
40.        // Constructor is 'protected'
41.        protected Singleton()
42.        {
43.        }
44.
45.        public static Singleton Instance()
46.        {
47.            // Uses lazy initialization.
48.            // Note: this is not thread safe.
49.            if (_instance == null)
50.            {
51.                _instance = new Singleton();

```

```

52.     }
53.
54.     return _instance;
55. }
56. }
57. }
58.
59.
60.
61.

```

Output

```
Objects are the same instance
```

Real-world code in C#

This **real-world** code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go through the one object that has knowledge about the state of the (web) farm.

```

1.
2.
3. using System;
4. using System.Collections.Generic;
5. using System.Threading;
6.
7. namespace DoFactory.GangOfFour.Singleton.RealWorld
8. {
9.     /// <summary>
10.    /// MainApp startup class for Real-World
11.    /// Singleton Design Pattern.
12.    /// </summary>
13.    class MainApp
14.    {
15.        /// <summary>
16.        /// Entry point into console application.
17.        /// </summary>
18.        static void Main()

```

```

19.     {
20.         LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
21.         LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
22.         LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
23.         LoadBalancer b4 = LoadBalancer.GetLoadBalancer();
24.
25.         // Same instance?
26.         if (b1 == b2 && b2 == b3 && b3 == b4)
27.         {
28.             Console.WriteLine("Same instance\n");
29.         }
30.
31.         // Load balance 15 server requests
32.         LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
33.         for (int i = 0; i < 15; i++)
34.         {
35.             string server = balancer.Server;
36.             Console.WriteLine("Dispatch Request to: " + server);
37.         }
38.
39.         // Wait for user
40.         Console.ReadKey();
41.     }
42. }
43.
44. /// <summary>
45. /// The 'Singleton' class
46. /// </summary>
47. class LoadBalancer
48. {
49.     private static LoadBalancer _instance;
50.     private List<string> _servers = new List<string>();
51.     private Random _random = new Random();
52.
53.     // Lock synchronization object
54.     private static object syncLock = new object();
55.
56.     // Constructor (protected)
57.     protected LoadBalancer()
58.     {
59.         // List of available servers
60.         _servers.Add("ServerI");
61.         _servers.Add("ServerII");
62.         _servers.Add("ServerIII");
63.         _servers.Add("ServerIV");
64.         _servers.Add("ServerV");
65.     }
66.
67.     public static LoadBalancer GetLoadBalancer()

```

```

68.     {
69.         // Support multithreaded applications through
70.         // 'Double checked locking' pattern which (once
71.         // the instance exists) avoids locking each
72.         // time the method is invoked
73.         if (_instance == null)
74.         {
75.             lock (syncLock)
76.             {
77.                 if (_instance == null)
78.                 {
79.                     _instance = new LoadBalancer();
80.                 }
81.             }
82.         }
83.
84.         return _instance;
85.     }
86.
87.     // Simple, but effective random load balancer
88.     public string Server
89.     {
90.         get
91.         {
92.             int r = _random.Next(_servers.Count);
93.             return _servers[r].ToString();
94.         }
95.     }
96. }
97. }
98.
99.
100.

```

Output

Same instance

```

ServerIII
ServerII
ServerI
ServerII
ServerI
ServerIII
ServerI
ServerIII
ServerIV
ServerII
ServerII
ServerIII

```

.NET Optimized code in C#

The [.NET optimized](#) code demonstrates the same code as above but uses more modern, built-in .NET features.

Here an elegant .NET specific solution is offered. The Singleton pattern simply uses a [private constructor](#) and a [static readonly](#) instance variable that is [lazily initialized](#). Thread safety is guaranteed by the compiler.

```
1.
2.
3. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Singleton.NETOptimized
7. {
8.     /// <summary>
9.     /// MainApp startup class for .NET optimized
10.    /// Singleton Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
20.            LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
21.            LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
22.            LoadBalancer b4 = LoadBalancer.GetLoadBalancer();
23.
24.            // Confirm these are the same instance
25.            if (b1 == b2 && b2 == b3 && b3 == b4)
26.            {
27.                Console.WriteLine("Same instance\n");
28.            }
29.
```

```

30.     // Next, load balance 15 requests for a server
31.     LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
32.     for (int i = 0; i < 15; i++)
33.     {
34.         string serverName = balancer.NextServer.Name;
35.         Console.WriteLine("Dispatch request to: " + serverName);
36.     }
37.
38.     // Wait for user
39.     Console.ReadKey();
40. }
41. }
42.
43. /// <summary>
44. /// The 'Singleton' class
45. /// </summary>
46. sealed class LoadBalancer
47. {
48.     // Static members are 'eagerly initialized', that is,
49.     // immediately when class is loaded for the first time.
50.     // .NET guarantees thread safety for static initialization
51.     private static readonly LoadBalancer _instance =
52.         new LoadBalancer();
53.
54.     // Type-safe generic list of servers
55.     private List<Server> _servers;
56.     private Random _random = new Random();
57.
58.     // Note: constructor is 'private'
59.     private LoadBalancer()
60.     {
61.         // Load list of available servers
62.         _servers = new List<Server>
63.         {
64.             new Server{ Name = "ServerI", IP = "120.14.220.18" },
65.             new Server{ Name = "ServerII", IP = "120.14.220.19" },
66.             new Server{ Name = "ServerIII", IP = "120.14.220.20" },
67.             new Server{ Name = "ServerIV", IP = "120.14.220.21" },
68.             new Server{ Name = "ServerV", IP = "120.14.220.22" },
69.         };
70.     }
71.
72.     public static LoadBalancer GetLoadBalancer()
73.     {
74.         return _instance;
75.     }
76.
77.     // Simple, but effective load balancer
78.     public Server NextServer

```

```

79.     {
80.         get
81.         {
82.             int r = _random.Next(_servers.Count);
83.             return _servers[r];
84.         }
85.     }
86. }
87.
88. /// <summary>
89. /// Represents a server machine
90. /// </summary>
91. class Server
92. {
93.     // Gets or sets server name
94.     public string Name { get; set; }
95.
96.     // Gets or sets server IP address
97.     public string IP { get; set; }
98. }
99. }
100.
101.
102.

```

Output

Same instance

```

ServerIV
ServerIV
ServerIII
ServerV
ServerII
ServerV
ServerII
ServerII
ServerI
ServerIV
ServerIV
ServerII
ServerI
ServerV
ServerIV

```


Abstract Factory

Definition

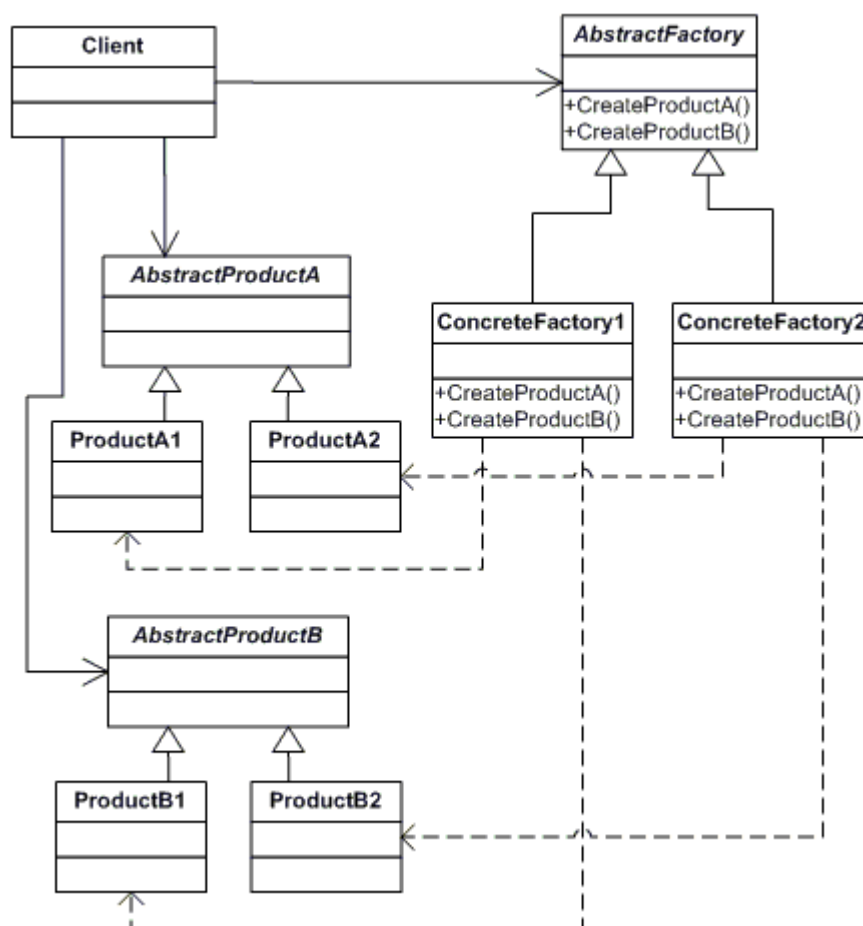
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Frequency of use:



High

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **AbstractFactory** (**ContinentFactory**)
 - declares an interface for operations that create abstract products
 - **ConcreteFactory** (**AfricaFactory, AmericaFactory**)
 - implements the operations to create concrete product objects
 - **AbstractProduct** (**Herbivore, Carnivore**)
 - declares an interface for a type of product object
 - **Product** (**Wildebeest, Lion, Bison, Wolf**)
 - defines a product object to be created by the corresponding concrete factory
 - implements the AbstractProduct interface
 - **Client** (**AnimalWorld**)
 - uses interfaces declared by AbstractFactory and AbstractProduct classes
-

Structural code in C#

This **structural** code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

```
1.
2.
3.
4. using System;
5.
6. namespace DoFactory.GangOfFour.Abstract.Structural
7. {
8.     /// <summary>
9.     /// MainApp startup class for Structural
10.    /// Abstract Factory Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
```

```

16.     /// </summary>
17.     public static void Main()
18.     {
19.         // Abstract factory #1
20.         AbstractFactory factory1 = new ConcreteFactory1();
21.         Client client1 = new Client(factory1);
22.         client1.Run();
23.
24.         // Abstract factory #2
25.         AbstractFactory factory2 = new ConcreteFactory2();
26.         Client client2 = new Client(factory2);
27.         client2.Run();
28.
29.         // Wait for user input
30.         Console.ReadKey();
31.     }
32. }
33.
34. /// <summary>
35. /// The 'AbstractFactory' abstract class
36. /// </summary>
37. abstract class AbstractFactory
38. {
39.     public abstract AbstractProductA CreateProductA();
40.     public abstract AbstractProductB CreateProductB();
41. }
42.
43.
44. /// <summary>
45. /// The 'ConcreteFactory1' class
46. /// </summary>
47. class ConcreteFactory1 : AbstractFactory
48. {
49.     public override AbstractProductA CreateProductA()
50.     {
51.         return new ProductA1();
52.     }
53.     public override AbstractProductB CreateProductB()
54.     {
55.         return new ProductB1();
56.     }
57. }
58.
59. /// <summary>
60. /// The 'ConcreteFactory2' class
61. /// </summary>
62. class ConcreteFactory2 : AbstractFactory
63. {
64.     public override AbstractProductA CreateProductA()

```

```

65.     {
66.         return new ProductA2();
67.     }
68.     public override AbstractProductB CreateProductB()
69.     {
70.         return new ProductB2();
71.     }
72. }
73.
74. /// <summary>
75. /// The 'AbstractProductA' abstract class
76. /// </summary>
77. abstract class AbstractProductA
78. {
79. }
80.
81. /// <summary>
82. /// The 'AbstractProductB' abstract class
83. /// </summary>
84. abstract class AbstractProductB
85. {
86.     public abstract void Interact(AbstractProductA a);
87. }
88.
89.
90. /// <summary>
91. /// The 'ProductA1' class
92. /// </summary>
93. class ProductA1 : AbstractProductA
94. {
95. }
96.
97. /// <summary>
98. /// The 'ProductB1' class
99. /// </summary>
100. class ProductB1 : AbstractProductB
101. {
102.     public override void Interact(AbstractProductA a)
103.     {
104.         Console.WriteLine(this.GetType().Name +
105.             " interacts with " + a.GetType().Name);
106.     }
107. }
108.
109. /// <summary>
110. /// The 'ProductA2' class
111. /// </summary>
112. class ProductA2 : AbstractProductA
113. {

```

```

114.     }
115.
116.     /// <summary>
117.     /// The 'ProductB2' class
118.     /// </summary>
119.     class ProductB2 : AbstractProductB
120.     {
121.         public override void Interact(AbstractProductA a)
122.         {
123.             Console.WriteLine(this.GetType().Name +
124.                 " interacts with " + a.GetType().Name);
125.         }
126.     }
127.
128.     /// <summary>
129.     /// The 'Client' class. Interaction environment for the products.
130.     /// </summary>
131.     class Client
132.     {
133.         private AbstractProductA _abstractProductA;
134.         private AbstractProductB _abstractProductB;
135.
136.         // Constructor
137.         public Client(AbstractFactory factory)
138.         {
139.             _abstractProductB = factory.CreateProductB();
140.             _abstractProductA = factory.CreateProductA();
141.         }
142.
143.         public void Run()
144.         {
145.             _abstractProductB.Interact(_abstractProductA);
146.         }
147.     }
148. }
149.
150.
151.
152.

```

Output

```

ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2

```

Real-world code in C#

This **real-world** code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

```
1.
2.
3. using System;
4.
5. namespace DoFactory.GangOfFour.Abstract.RealWorld
6. {
7.     /// <summary>
8.     /// MainApp startup class for Real-World
9.     /// Abstract Factory Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        public static void Main()
17.        {
18.            // Create and run the African animal world
19.            ContinentFactory africa = new AfricaFactory();
20.            AnimalWorld world = new AnimalWorld(africa);
21.            world.RunFoodChain();
22.
23.            // Create and run the American animal world
24.            ContinentFactory america = new AmericaFactory();
25.            world = new AnimalWorld(america);
26.            world.RunFoodChain();
27.
28.            // Wait for user input
29.            Console.ReadKey();
30.        }
31.    }
32.
33.
34.    /// <summary>
35.    /// The 'AbstractFactory' abstract class
36.    /// </summary>
37.    abstract class ContinentFactory
38.    {
```

```

39.     public abstract Herbivore CreateHerbivore();
40.     public abstract Carnivore CreateCarnivore();
41. }
42.
43. /// <summary>
44. /// The 'ConcreteFactory1' class
45. /// </summary>
46. class AfricaFactory : ContinentFactory
47. {
48.     public override Herbivore CreateHerbivore()
49.     {
50.         return new Wildebeest();
51.     }
52.     public override Carnivore CreateCarnivore()
53.     {
54.         return new Lion();
55.     }
56. }
57.
58. /// <summary>
59. /// The 'ConcreteFactory2' class
60. /// </summary>
61. class AmericaFactory : ContinentFactory
62. {
63.     public override Herbivore CreateHerbivore()
64.     {
65.         return new Bison();
66.     }
67.     public override Carnivore CreateCarnivore()
68.     {
69.         return new Wolf();
70.     }
71. }
72.
73. /// <summary>
74. /// The 'AbstractProductA' abstract class
75. /// </summary>
76. abstract class Herbivore
77. {
78. }
79.
80. /// <summary>
81. /// The 'AbstractProductB' abstract class
82. /// </summary>
83. abstract class Carnivore
84. {
85.     public abstract void Eat(Herbivore h);
86. }
87.

```

```

88.  /// <summary>
89.  /// The 'ProductA1' class
90.  /// </summary>
91.  class Wildebeest : Herbivore
92.  {
93.  }
94.
95.  /// <summary>
96.  /// The 'ProductB1' class
97.  /// </summary>
98.  class Lion : Carnivore
99.  {
100.      public override void Eat(Herbivore h)
101.      {
102.          // Eat Wildebeest
103.          Console.WriteLine(this.GetType().Name +
104.              " eats " + h.GetType().Name);
105.      }
106.  }
107.
108.  /// <summary>
109.  /// The 'ProductA2' class
110.  /// </summary>
111.  class Bison : Herbivore
112.  {
113.  }
114.
115.  /// <summary>
116.  /// The 'ProductB2' class
117.  /// </summary>
118.  class Wolf : Carnivore
119.  {
120.      public override void Eat(Herbivore h)
121.      {
122.          // Eat Bison
123.          Console.WriteLine(this.GetType().Name +
124.              " eats " + h.GetType().Name);
125.      }
126.  }
127.
128.  /// <summary>
129.  /// The 'Client' class
130.  /// </summary>
131.  class AnimalWorld
132.  {
133.      private Herbivore _herbivore;
134.      private Carnivore _carnivore;
135.
136.      // Constructor

```



```

137.         public AnimalWorld(ContinentFactory factory)
138.         {
139.             _carnivore = factory.CreateCarnivore();
140.             _herbivore = factory.CreateHerbivore();
141.         }
142.
143.         public void RunFoodChain()
144.         {
145.             _carnivore.Eat(_herbivore);
146.         }
147.     }
148. }
149.
150.
151.

```

Output

```

Lion eats Wildebeest
Wolf eats Bison

```

Structural Patterns

Adapter

Definition

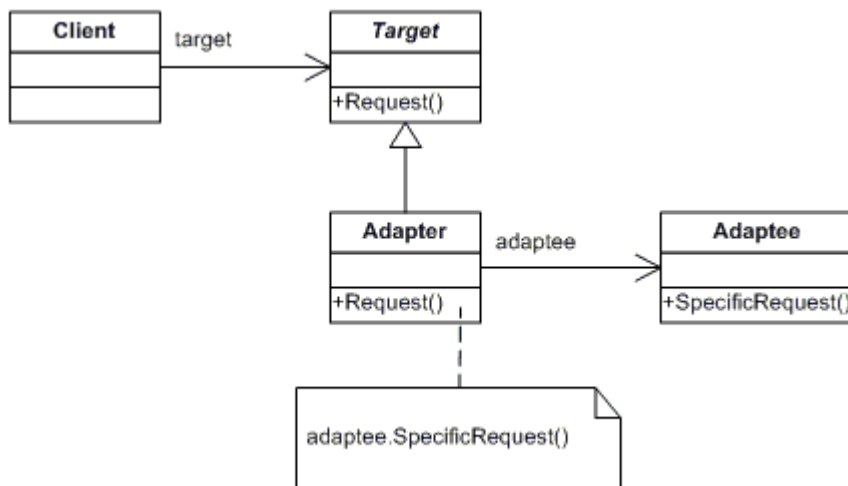
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Frequency of use:



Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Target** (**ChemicalCompound**)
 - defines the domain-specific interface that Client uses.
- **Adapter** (**Compound**)
 - adapts the interface Adaptee to the Target interface.
- **Adaptee** (**ChemicalDatabank**)
 - defines an existing interface that needs adapting.
- **Client** (**AdapterApp**)
 - collaborates with objects conforming to the Target interface.

Structural code in C#

This **structural** code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

```

1.
2.
3. using System;
4.
5. namespace DoFactory.GangOfFour.Adapter.Structural
6. {
7.     /// <summary>
8.     /// MainApp startup class for Structural
9.     /// Adapter Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        static void Main()
17.        {
18.            // Create adapter and place a request
19.            Target target = new Adapter();
20.            target.Request();
21.
22.            // Wait for user
23.            Console.ReadKey();
24.        }
25.    }
26.
27.    /// <summary>
28.    /// The 'Target' class
29.    /// </summary>
30.    class Target
31.    {
32.        public virtual void Request()
33.        {
34.            Console.WriteLine("Called Target Request()");
35.        }
36.    }
37.
38.    /// <summary>
39.    /// The 'Adapter' class
40.    /// </summary>
41.    class Adapter : Target
42.    {
43.        private Adaptee _adaptee = new Adaptee();
44.
45.        public override void Request()
46.        {
47.            // Possibly do some other work
48.            // and then call SpecificRequest
49.            _adaptee.SpecificRequest();
50.        }

```

```

51. }
52.
53. /// <summary>
54. /// The 'Adaptee' class
55. /// </summary>
56. class Adaptee
57. {
58.     public void SpecificRequest()
59.     {
60.         Console.WriteLine("Called SpecificRequest()");
61.     }
62. }
63. }
64.
65.
66.
67.

```

Output

```
Called SpecificRequest()
```

Real-world code in C#

This **real-world** code demonstrates the use of a legacy chemical databank. Chemical compound objects access the databank through an Adapter interface.

```

1.
2.
3. using System;
4.
5. namespace DoFactory.GangOfFour.Adapter.RealWorld
6. {
7.     /// <summary>
8.     /// MainApp startup class for Real-World
9.     /// Adapter Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {

```

```

13.    /// <summary>
14.    /// Entry point into console application.
15.    /// </summary>
16.    static void Main()
17.    {
18.        // Non-adapted chemical compound
19.        Compound unknown = new Compound("Unknown");
20.        unknown.Display();
21.
22.        // Adapted chemical compounds
23.        Compound water = new RichCompound("Water");
24.        water.Display();
25.
26.        Compound benzene = new RichCompound("Benzene");
27.        benzene.Display();
28.
29.        Compound ethanol = new RichCompound("Ethanol");
30.        ethanol.Display();
31.
32.        // Wait for user
33.        Console.ReadKey();
34.    }
35. }
36.
37.    /// <summary>
38.    /// The 'Target' class
39.    /// </summary>
40.    class Compound
41.    {
42.        protected string _chemical;
43.        protected float _boilingPoint;
44.        protected float _meltingPoint;
45.        protected double _molecularWeight;
46.        protected string _molecularFormula;
47.
48.        // Constructor
49.        public Compound(string chemical)
50.        {
51.            this._chemical = chemical;
52.        }
53.
54.        public virtual void Display()
55.        {
56.            Console.WriteLine("\nCompound: {0} ----- ", _chemical);
57.        }
58.    }
59.
60.    /// <summary>
61.    /// The 'Adapter' class

```

```

62.  /// </summary>
63.  class RichCompound : Compound
64.  {
65.      private ChemicalDatabank _bank;
66.
67.      // Constructor
68.      public RichCompound(string name)
69.          : base(name)
70.      {
71.      }
72.
73.      public override void Display()
74.      {
75.          // The Adaptee
76.          _bank = new ChemicalDatabank();
77.
78.          _boilingPoint = _bank.GetCriticalPoint(_chemical, "B");
79.          _meltingPoint = _bank.GetCriticalPoint(_chemical, "M");
80.          _molecularWeight = _bank.GetMolecularWeight(_chemical);
81.          _molecularFormula = _bank.GetMolecularStructure(_chemical);
82.
83.          base.Display();
84.          Console.WriteLine(" Formula: {0}", _molecularFormula);
85.          Console.WriteLine(" Weight : {0}", _molecularWeight);
86.          Console.WriteLine(" Melting Pt: {0}", _meltingPoint);
87.          Console.WriteLine(" Boiling Pt: {0}", _boilingPoint);
88.      }
89.  }
90.
91.  /// <summary>
92.  /// The 'Adaptee' class
93.  /// </summary>
94.  class ChemicalDatabank
95.  {
96.      // The databank 'legacy API'
97.      public float GetCriticalPoint(string compound, string point)
98.      {
99.          // Melting Point
100.         if (point == "M")
101.         {
102.             switch (compound.ToLower())
103.             {
104.                 case "water": return 0.0f;
105.                 case "benzene": return 5.5f;
106.                 case "ethanol": return -114.1f;
107.                 default: return 0f;
108.             }
109.         }
110.         // Boiling Point

```

```

111.         else
112.         {
113.             switch (compound.ToLower())
114.             {
115.                 case "water": return 100.0f;
116.                 case "benzene": return 80.1f;
117.                 case "ethanol": return 78.3f;
118.                 default: return 0f;
119.             }
120.         }
121.     }
122.
123.     public string GetMolecularStructure(string compound)
124.     {
125.         switch (compound.ToLower())
126.         {
127.             case "water": return "H2O";
128.             case "benzene": return "C6H6";
129.             case "ethanol": return "C2H5OH";
130.             default: return "";
131.         }
132.     }
133.
134.     public double GetMolecularWeight(string compound)
135.     {
136.         switch (compound.ToLower())
137.         {
138.             case "water": return 18.015;
139.             case "benzene": return 78.1134;
140.             case "ethanol": return 46.0688;
141.             default: return 0d;
142.         }
143.     }
144. }
145.
146.
147.
148.
149.

```

Output

Compound: Unknown -----

Compound: Water -----

Formula: H2O

Weight : 18.015

Melting Pt: 0

```
Boiling Pt: 100
```

```
Compound: Benzene -----
```

```
Formula: C6H6
```

```
Weight : 78.1134
```

```
Melting Pt: 5.5
```

```
Boiling Pt: 80.1
```

```
Compound: Alcohol -----
```

```
Formula: C2H6O2
```

```
Weight : 46.0688
```

```
Melting Pt: -114.1
```

```
Boiling Pt: 78.3
```

Composite

Definition

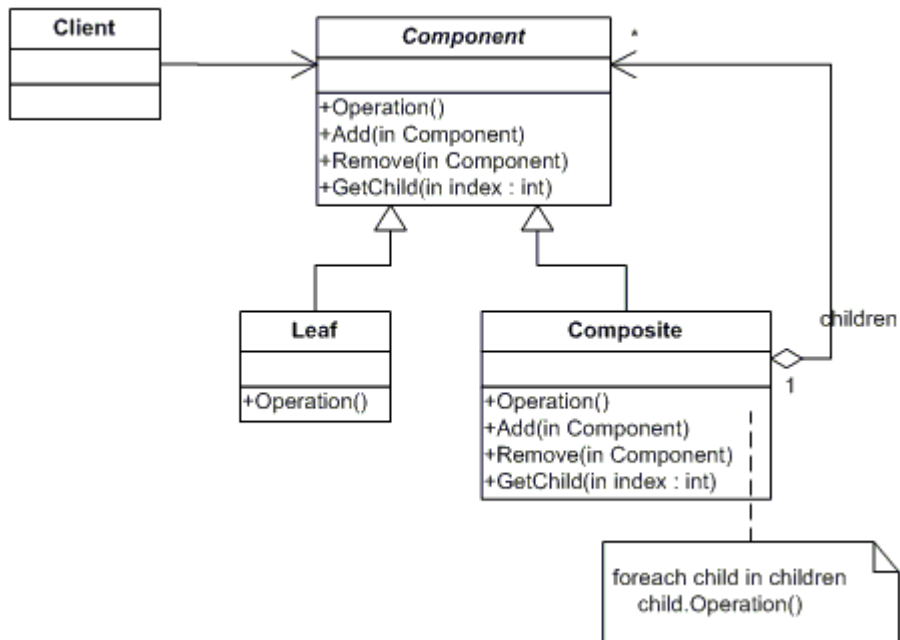
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Frequency of use:



Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Component (DrawingElement)**
 - declares the interface for objects in the composition.
 - implements default behavior for the interface common to all classes, as appropriate.
 - declares an interface for accessing and managing its child components.
 - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf (PrimitiveElement)**
 - represents leaf objects in the composition. A leaf has no children.
 - defines behavior for primitive objects in the composition.
- **Composite (CompositeElement)**
 - defines behavior for components having children.
 - stores child components.
 - implements child-related operations in the Component interface.
- **Client (CompositeApp)**
 - manipulates objects in the composition through the Component interface.

Structural code in C#

This **structural** code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

```
1.
2.
3. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Composite.Structural
7. {
8.     /// <summary>
9.     /// MainApp startup class for Structural
10.    /// Composite Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            // Create a tree structure
20.            Composite root = new Composite("root");
21.            root.Add(new Leaf("Leaf A"));
22.            root.Add(new Leaf("Leaf B"));
23.
24.            Composite comp = new Composite("Composite X");
25.            comp.Add(new Leaf("Leaf XA"));
26.            comp.Add(new Leaf("Leaf XB"));
27.
28.            root.Add(comp);
29.            root.Add(new Leaf("Leaf C"));
30.
31.            // Add and remove a leaf
32.            Leaf leaf = new Leaf("Leaf D");
33.            root.Add(leaf);
34.            root.Remove(leaf);
35.
36.            // Recursively display tree
37.            root.Display(1);
38.
39.            // Wait for user
40.            Console.ReadKey();
```

```

41.     }
42. }
43.
44. /// <summary>
45. /// The 'Component' abstract class
46. /// </summary>
47. abstract class Component
48. {
49.     protected string name;
50.
51.     // Constructor
52.     public Component(string name)
53.     {
54.         this.name = name;
55.     }
56.
57.     public abstract void Add(Component c);
58.     public abstract void Remove(Component c);
59.     public abstract void Display(int depth);
60. }
61.
62. /// <summary>
63. /// The 'Composite' class
64. /// </summary>
65. class Composite : Component
66. {
67.     private List<Component> _children = new List<Component>();
68.
69.     // Constructor
70.     public Composite(string name)
71.         : base(name)
72.     {
73.     }
74.
75.     public override void Add(Component component)
76.     {
77.         _children.Add(component);
78.     }
79.
80.     public override void Remove(Component component)
81.     {
82.         _children.Remove(component);
83.     }
84.
85.     public override void Display(int depth)
86.     {
87.         Console.WriteLine(new String('-', depth) + name);
88.
89.         // Recursively display child nodes

```

```

90.     foreach (Component component in _children)
91.     {
92.         component.Display(depth + 2);
93.     }
94. }
95. }
96.
97. /// <summary>
98. /// The 'Leaf' class
99. /// </summary>
100.    class Leaf : Component
101.    {
102.        // Constructor
103.        public Leaf(string name)
104.            : base(name)
105.        {
106.        }
107.
108.        public override void Add(Component c)
109.        {
110.            Console.WriteLine("Cannot add to a leaf");
111.        }
112.
113.        public override void Remove(Component c)
114.        {
115.            Console.WriteLine("Cannot remove from a leaf");
116.        }
117.
118.        public override void Display(int depth)
119.        {
120.            Console.WriteLine(new String('-', depth) + name);
121.        }
122.    }
123. }
124.
125.
126.
127.

```

Output

```

-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C

```

Real-world code in C#

This **real-world** code demonstrates the Composite pattern used in building a graphical tree structure made up of primitive nodes (lines, circles, etc) and composite nodes (groups of drawing elements that make up more complex elements).

```
1.
2.
3. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Composite.RealWorld
7. {
8.     /// <summary>
9.     /// MainApp startup class for Real-World
10.    /// Composite Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            // Create a tree structure
20.            CompositeElement root =
21.                new CompositeElement("Picture");
22.            root.Add(new PrimitiveElement("Red Line"));
23.            root.Add(new PrimitiveElement("Blue Circle"));
24.            root.Add(new PrimitiveElement("Green Box"));
25.
26.            // Create a branch
27.            CompositeElement comp =
28.                new CompositeElement("Two Circles");
29.            comp.Add(new PrimitiveElement("Black Circle"));
30.            comp.Add(new PrimitiveElement("White Circle"));
31.            root.Add(comp);
32.
33.            // Add and remove a PrimitiveElement
34.            PrimitiveElement pe =
```

```

35.         new PrimitiveElement("Yellow Line");
36.     root.Add(pe);
37.     root.Remove(pe);
38.
39.     // Recursively display nodes
40.     root.Display(1);
41.
42.     // Wait for user
43.     Console.ReadKey();
44. }
45. }
46.
47. /// <summary>
48. /// The 'Component' Treenode
49. /// </summary>
50. abstract class DrawingElement
51. {
52.     protected string _name;
53.
54.     // Constructor
55.     public DrawingElement(string name)
56.     {
57.         this._name = name;
58.     }
59.
60.     public abstract void Add(DrawingElement d);
61.     public abstract void Remove(DrawingElement d);
62.     public abstract void Display(int indent);
63. }
64.
65. /// <summary>
66. /// The 'Leaf' class
67. /// </summary>
68. class PrimitiveElement : DrawingElement
69. {
70.     // Constructor
71.     public PrimitiveElement(string name)
72.         : base(name)
73.     {
74.     }
75.
76.     public override void Add(DrawingElement c)
77.     {
78.         Console.WriteLine(
79.             "Cannot add to a PrimitiveElement");
80.     }
81.
82.     public override void Remove(DrawingElement c)
83.     {

```

```

84.     Console.WriteLine(
85.         "Cannot remove from a PrimitiveElement");
86.     }
87.
88.     public override void Display(int indent)
89.     {
90.         Console.WriteLine(
91.             new String('-', indent) + " " + _name);
92.     }
93. }
94.
95. /// <summary>
96. /// The 'Composite' class
97. /// </summary>
98. class CompositeElement : DrawingElement
99. {
100.     private List<DrawingElement> elements =
101.         new List<DrawingElement>();
102.
103.     // Constructor
104.     public CompositeElement(string name)
105.         : base(name)
106.     {
107.     }
108.
109.     public override void Add(DrawingElement d)
110.     {
111.         elements.Add(d);
112.     }
113.
114.     public override void Remove(DrawingElement d)
115.     {
116.         elements.Remove(d);
117.     }
118.
119.     public override void Display(int indent)
120.     {
121.         Console.WriteLine(new String('-', indent) +
122.             "+ " + _name);
123.
124.         // Display each child element on this node
125.         foreach (DrawingElement d in elements)
126.         {
127.             d.Display(indent + 2);
128.         }
129.     }
130. }
131. }
132.

```

133.
134.
135.

Output

```
--+ Picture  
--- Red Line  
--- Blue Circle  
--- Green Box  
----+ Two Circles  
----- Black Circle  
----- White Circle
```

Behavioral Patterns

Command

Definition

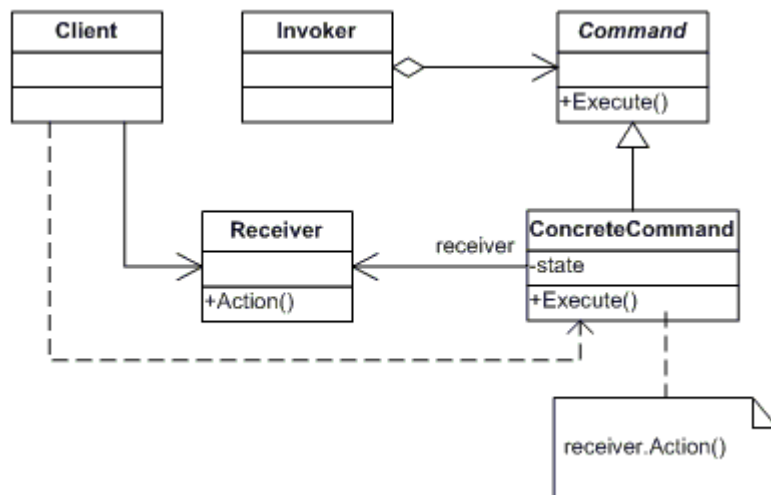
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Frequency of use:



Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Command (Command)**
 - declares an interface for executing an operation
- **ConcreteCommand (CalculatorCommand)**
 - defines a binding between a Receiver object and an action
 - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (CommandApp)**
 - creates a ConcreteCommand object and sets its receiver
- **Invoker (User)**
 - asks the command to carry out the request
- **Receiver (Calculator)**
 - knows how to perform the operations associated with carrying out the request.

Structural code in C#

This **structural** code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.

```
1.
2.
3. using System;
4.
5. namespace DoFactory.GangOfFour.Command.Structural
6. {
7.     /// <summary>
8.     /// MainApp startup class for Structural
9.     /// Command Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        static void Main()
17.        {
18.            // Create receiver, command, and invoker
19.            Receiver receiver = new Receiver();
20.            Command command = new ConcreteCommand(receiver);
21.            Invoker invoker = new Invoker();
22.
23.            // Set and execute command
24.            invoker.SetCommand(command);
25.            invoker.ExecuteCommand();
26.
27.            // Wait for user
28.            Console.ReadKey();
29.        }
30.    }
31.
32.    /// <summary>
33.    /// The 'Command' abstract class
34.    /// </summary>
35.    abstract class Command
36.    {
37.        protected Receiver receiver;
38.
39.        // Constructor
40.        public Command(Receiver receiver)
41.        {
42.            this.receiver = receiver;
43.        }
44.
45.        public abstract void Execute();
46.    }
47.
```

```

48.  /// <summary>
49.  /// The 'ConcreteCommand' class
50.  /// </summary>
51.  class ConcreteCommand : Command
52.  {
53.      // Constructor
54.      public ConcreteCommand(Receiver receiver) :
55.          base(receiver)
56.      {
57.      }
58.
59.      public override void Execute()
60.      {
61.          receiver.Action();
62.      }
63.  }
64.
65.  /// <summary>
66.  /// The 'Receiver' class
67.  /// </summary>
68.  class Receiver
69.  {
70.      public void Action()
71.      {
72.          Console.WriteLine("Called Receiver.Action()");
73.      }
74.  }
75.
76.  /// <summary>
77.  /// The 'Invoker' class
78.  /// </summary>
79.  class Invoker
80.  {
81.      private Command _command;
82.
83.      public void SetCommand(Command command)
84.      {
85.          this._command = command;
86.      }
87.
88.      public void ExecuteCommand()
89.      {
90.          _command.Execute();
91.      }
92.  }
93. }
94.
95.
96.

```

Output

```
Called Receiver.Action()
```

Real-world code in C#

This **real-world** code demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C# the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

```

1.
2.
3. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Command.RealWorld
7. {
8.     /// <summary>
9.     /// MainApp startup class for Real-World
10.    /// Command Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            // Create user and let her compute
20.            User user = new User();
21.
22.            // User presses calculator buttons
23.            user.Compute('+', 100);
24.            user.Compute('-', 50);
25.            user.Compute('*', 10);
26.            user.Compute('/', 2);
27.

```

```

28.         // Undo 4 commands
29.         user.Undo(4);
30.
31.         // Redo 3 commands
32.         user.Redo(3);
33.
34.         // Wait for user
35.         Console.ReadKey();
36.     }
37. }
38.
39. /// <summary>
40. /// The 'Command' abstract class
41. /// </summary>
42. abstract class Command
43. {
44.     public abstract void Execute();
45.     public abstract void UnExecute();
46. }
47.
48. /// <summary>
49. /// The 'ConcreteCommand' class
50. /// </summary>
51. class CalculatorCommand : Command
52. {
53.     private char _operator;
54.     private int _operand;
55.     private Calculator _calculator;
56.
57.     // Constructor
58.     public CalculatorCommand(Calculator calculator,
59.         char @operator, int operand)
60.     {
61.         this._calculator = calculator;
62.         this._operator = @operator;
63.         this._operand = operand;
64.     }
65.
66.     // Gets operator
67.     public char Operator
68.     {
69.         set { _operator = value; }
70.     }
71.
72.     // Get operand
73.     public int Operand
74.     {
75.         set { _operand = value; }
76.     }

```

```

77.
78.     // Execute new command
79.     public override void Execute()
80.     {
81.         _calculator.Operation(_operator, _operand);
82.     }
83.
84.     // Unexecute last command
85.     public override void UnExecute()
86.     {
87.         _calculator.Operation(Undo(_operator), _operand);
88.     }
89.
90.     // Returns opposite operator for given operator
91.     private char Undo(char @operator)
92.     {
93.         switch (@operator)
94.         {
95.             case '+': return '-';
96.             case '-': return '+';
97.             case '*': return '/';
98.             case '/': return '*';
99.             default: throw new
100.                 ArgumentException("@operator");
101.         }
102.     }
103. }
104.
105.     /// <summary>
106.     /// The 'Receiver' class
107.     /// </summary>
108.     class Calculator
109.     {
110.         private int _curr = 0;
111.
112.         public void Operation(char @operator, int operand)
113.         {
114.             switch (@operator)
115.             {
116.                 case '+': _curr += operand; break;
117.                 case '-': _curr -= operand; break;
118.                 case '*': _curr *= operand; break;
119.                 case '/': _curr /= operand; break;
120.             }
121.             Console.WriteLine(
122.                 "Current value = {0,3} (following {1} {2})",
123.                 _curr, @operator, operand);
124.         }
125.     }

```

```

126.
127.     /// <summary>
128.     /// The 'Invoker' class
129.     /// </summary>
130.     class User
131.     {
132.         // Initializers
133.         private Calculator _calculator = new Calculator();
134.         private List<Command> _commands = new List<Command>();
135.         private int _current = 0;
136.
137.         public void Redo(int levels)
138.         {
139.             Console.WriteLine("\n---- Redo {0} levels ", levels);
140.             // Perform redo operations
141.             for (int i = 0; i < levels; i++)
142.             {
143.                 if (_current < _commands.Count - 1)
144.                 {
145.                     Command command = _commands[_current++];
146.                     command.Execute();
147.                 }
148.             }
149.         }
150.
151.         public void Undo(int levels)
152.         {
153.             Console.WriteLine("\n---- Undo {0} levels ", levels);
154.             // Perform undo operations
155.             for (int i = 0; i < levels; i++)
156.             {
157.                 if (_current > 0)
158.                 {
159.                     Command command = _commands[--_current] as Command;
160.                     command.UnExecute();
161.                 }
162.             }
163.         }
164.
165.         public void Compute(char @operator, int operand)
166.         {
167.             // Create command operation and execute it
168.             Command command = new CalculatorCommand(
169.                 _calculator, @operator, operand);
170.             command.Execute();
171.
172.             // Add command to undo list
173.             _commands.Add(command);
174.             _current++;

```

```

175.         }
176.     }
177. }
178.
179.
180.
181.

```

Output

```

Current value = 100 (following + 100)
Current value = 50 (following - 50)
Current value = 500 (following * 10)
Current value = 250 (following / 2)

---- Undo 4 levels
Current value = 500 (following * 2)
Current value = 50 (following / 10)
Current value = 100 (following + 50)
Current value = 0 (following - 100)

---- Redo 3 levels
Current value = 100 (following + 100)
Current value = 50 (following - 50)
Current value = 500 (following * 10)

```

Interpreter

Definition

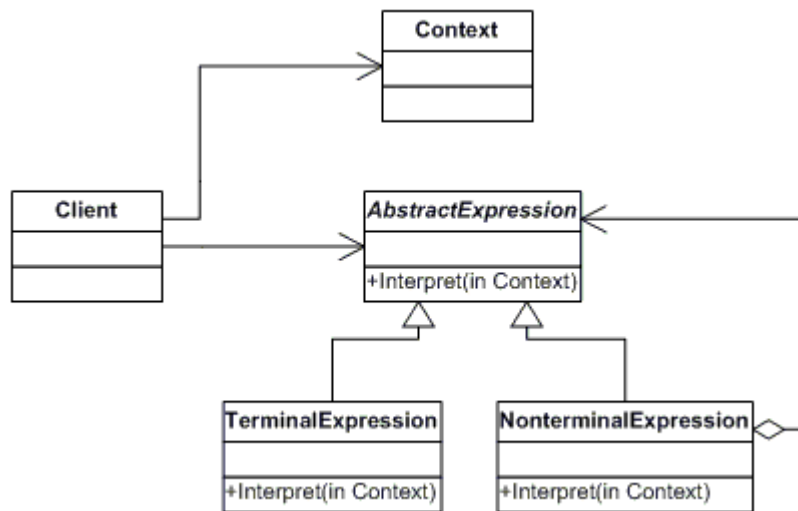
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Frequency of use:



Low

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **AbstractExpression (Expression)**
 - declares an interface for executing an operation
- **TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)**
 - implements an Interpret operation associated with terminal symbols in the grammar.
 - an instance is required for every terminal symbol in the sentence.
- **NonterminalExpression (not used)**
 - one such class is required for every rule $R ::= R_1 R_2 \dots R_n$ in the grammar
 - maintains instance variables of type **AbstractExpression** for each of the symbols R_1 through R_n .
 - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n .
- **Context (Context)**
 - contains information that is global to the interpreter
- **Client (InterpreterApp)**
 - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the **NonterminalExpression** and **TerminalExpression** classes
 - invokes the Interpret operation

Structural code in C#

This **structural** code demonstrates the Interpreter patterns, which using a defined grammar, provides the interpreter that processes parsed statements.

```
1.
2.
3. using System;
4. using System.Collections;
5.
6. namespace DoFactory.GangOfFour.Interpreter.Structural
7. {
8.     /// <summary>
9.     /// MainApp startup class for Structural
10.    /// Interpreter Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            Context context = new Context();
20.
21.            // Usually a tree
22.            ArrayList list = new ArrayList();
23.
24.            // Populate 'abstract syntax tree'
25.            list.Add(new TerminalExpression());
26.            list.Add(new NonterminalExpression());
27.            list.Add(new TerminalExpression());
28.            list.Add(new TerminalExpression());
29.
30.            // Interpret
31.            foreach (AbstractExpression exp in list)
32.            {
33.                exp.Interpret(context);
34.            }
35.
36.            // Wait for user
37.            Console.ReadKey();
38.        }
39.    }
40.
```

```

41.  /// <summary>
42.  /// The 'Context' class
43.  /// </summary>
44.  class Context
45.  {
46.  }
47.
48.  /// <summary>
49.  /// The 'AbstractExpression' abstract class
50.  /// </summary>
51.  abstract class AbstractExpression
52.  {
53.      public abstract void Interpret(Context context);
54.  }
55.
56.  /// <summary>
57.  /// The 'TerminalExpression' class
58.  /// </summary>
59.  class TerminalExpression : AbstractExpression
60.  {
61.      public override void Interpret(Context context)
62.      {
63.          Console.WriteLine("Called Terminal.Interpret()");
64.      }
65.  }
66.
67.  /// <summary>
68.  /// The 'NonterminalExpression' class
69.  /// </summary>
70.  class NonterminalExpression : AbstractExpression
71.  {
72.      public override void Interpret(Context context)
73.      {
74.          Console.WriteLine("Called Nonterminal.Interpret()");
75.      }
76.  }
77. }
78.
79.
80.
81.

```

Output

```

Called Terminal.Interpret()
Called Nonterminal.Interpret()
Called Terminal.Interpret()
Called Terminal.Interpret()

```

Real-world code in C#

This **real-world** code demonstrates the Interpreter pattern which is used to convert a Roman numeral to a decimal.

```
1.
2.
3. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Interpreter.RealWorld
7. {
8.     /// <summary>
9.     /// MainApp startup class for Real-World
10.    /// Interpreter Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            string roman = "MCMXXVIII";
20.            Context context = new Context(roman);
21.
22.            // Build the 'parse tree'
23.            List<Expression> tree = new List<Expression>();
24.            tree.Add(new ThousandExpression());
25.            tree.Add(new HundredExpression());
26.            tree.Add(new TenExpression());
27.            tree.Add(new OneExpression());
28.
29.            // Interpret
30.            foreach (Expression exp in tree)
31.            {
32.                exp.Interpret(context);
33.            }
34.
35.            Console.WriteLine("{0} = {1}",
```

```

36.         roman, context.Output);
37.
38.         // Wait for user
39.         Console.ReadKey();
40.     }
41. }
42.
43. /// <summary>
44. /// The 'Context' class
45. /// </summary>
46. class Context
47. {
48.     private string _input;
49.     private int _output;
50.
51.     // Constructor
52.     public Context(string input)
53.     {
54.         this._input = input;
55.     }
56.
57.     // Gets or sets input
58.     public string Input
59.     {
60.         get { return _input; }
61.         set { _input = value; }
62.     }
63.
64.     // Gets or sets output
65.     public int Output
66.     {
67.         get { return _output; }
68.         set { _output = value; }
69.     }
70. }
71.
72. /// <summary>
73. /// The 'AbstractExpression' class
74. /// </summary>
75. abstract class Expression
76. {
77.     public void Interpret(Context context)
78.     {
79.         if (context.Input.Length == 0)
80.             return;
81.
82.         if (context.Input.StartsWith(Nine()))
83.         {
84.             context.Output += (9 * Multiplier());

```

```

85.         context.Input = context.Input.Substring(2);
86.     }
87.     else if (context.Input.StartsWith(Four()))
88.     {
89.         context.Output += (4 * Multiplier());
90.         context.Input = context.Input.Substring(2);
91.     }
92.     else if (context.Input.StartsWith(Five()))
93.     {
94.         context.Output += (5 * Multiplier());
95.         context.Input = context.Input.Substring(1);
96.     }
97.
98.     while (context.Input.StartsWith(One()))
99.     {
100.         context.Output += (1 * Multiplier());
101.         context.Input = context.Input.Substring(1);
102.     }
103. }
104.
105.     public abstract string One();
106.     public abstract string Four();
107.     public abstract string Five();
108.     public abstract string Nine();
109.     public abstract int Multiplier();
110. }
111.
112.     /// <summary>
113.     /// A 'TerminalExpression' class
114.     /// <remarks>
115.     /// Thousand checks for the Roman Numeral M
116.     /// </remarks>
117.     /// </summary>
118.     class ThousandExpression : Expression
119.     {
120.         public override string One() { return "M"; }
121.         public override string Four() { return " "; }
122.         public override string Five() { return " "; }
123.         public override string Nine() { return " "; }
124.         public override int Multiplier() { return 1000; }
125.     }
126.
127.     /// <summary>
128.     /// A 'TerminalExpression' class
129.     /// <remarks>
130.     /// Hundred checks C, CD, D or CM
131.     /// </remarks>
132.     /// </summary>
133.     class HundredExpression : Expression

```

```

134.         {
135.             public override string One() { return "C"; }
136.             public override string Four() { return "CD"; }
137.             public override string Five() { return "D"; }
138.             public override string Nine() { return "CM"; }
139.             public override int Multiplier() { return 100; }
140.         }
141.
142.         /// <summary>
143.         /// A 'TerminalExpression' class
144.         /// <remarks>
145.         /// Ten checks for X, XL, L and XC
146.         /// </remarks>
147.         /// </summary>
148.         class TenExpression : Expression
149.         {
150.             public override string One() { return "X"; }
151.             public override string Four() { return "XL"; }
152.             public override string Five() { return "L"; }
153.             public override string Nine() { return "XC"; }
154.             public override int Multiplier() { return 10; }
155.         }
156.
157.         /// <summary>
158.         /// A 'TerminalExpression' class
159.         /// <remarks>
160.         /// One checks for I, II, III, IV, V, VI, VI, VII, VIII, IX
161.         /// </remarks>
162.         /// </summary>
163.         class OneExpression : Expression
164.         {
165.             public override string One() { return "I"; }
166.             public override string Four() { return "IV"; }
167.             public override string Five() { return "V"; }
168.             public override string Nine() { return "IX"; }
169.             public override int Multiplier() { return 1; }
170.         }
171.     }
172.
173.
174.

```

Output

```
MCMXXVIII = 1928
```