JAIN DEEMED-TO-BE UNIVERSITY

SCHOOL OF COMPUTER APPLICATIONS & INFORMATION TECHNOLOGY

Department of Master of Computer Applications

Semester III

C# & .NET Technologies

Unit III

Collections, Delegates and Events

Collections

- C# also includes specialized classes that hold many values or objects in a specific series, that are called 'collection'.
- C# collection types are designed to store, manage and manipulate similar data more efficiently.
- Collection types implement the following common functionality:
  - Adding and inserting items to a collection
  - Removing items from a collection
  - Finding, sorting, searching items
  - Replacing items
  - Copy and clone collections and items
  - Capacity and Count properties to find the capacity of the collection and number of items in the collection

Collections

- There are two types of collections available in C#: non-generic collections and generic collections.

Non-Generic	Similar Generic Type
ArrayList	List<T>
Hashtable	Dictionary<TKey,TValue>
SortedList	SortedList<TKey,TValue>
Queue	Queue<T>
Stack	Stack<T>
IEnumerable	IEnumerable<T>
ICollection	N/A (use IEnumerable<T> anything that extends it)
N/A	ICollection<T>
IList	IList<T>
CollectionBase	Collection<T>
ReadOnlyCollectionBase	ReadOnlyCollection<T>
DictionaryBase	N/A (just implement IDictionary<TKey,TValue>
N/A	SortedDictionary<TKey,TValue>
N/A	KeyedCollection<TKey,TItem>
N/A	LinkedList<T>

difference between generic and non-generic

Generic vs Non-Generic Collection in C#	
A Generic collection is a class that provides type safety without having to derive from a base collection type and implement type-specific members.	A Non-generic collection is a specialized class for data storage and retrieval that provides support for stacks, queues, lists and hash tables.
Namespace	
The Generic Collection classes are in the System. Collections. Generics namespace.	The Non -generic Collection classes are in the System. Collections namespace.
Type	
A Generic Collection is strongly typed.	A Non-Generic Collection is not strongly typed.
Storing Elements	
The Generic Collections store elements internally in arrays of their actual types.	The Non-generic collections store elements internally in object arrays so it can store any type of data.

difference between generic and non-generic

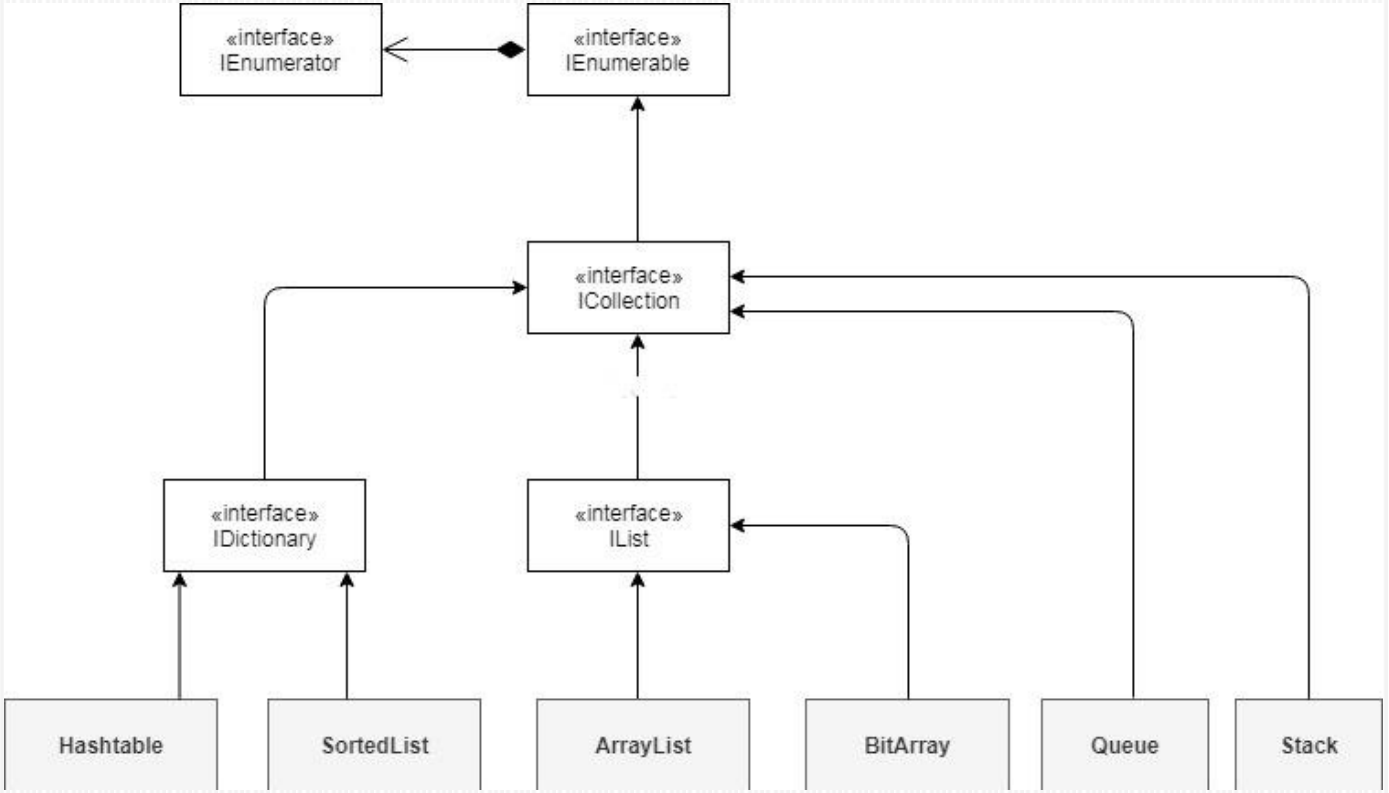
- **Non-Generic collections** – These are the collections that can hold elements of different data types. It holds all elements as object type. So it includes overhead of type conversions (overhead of implicit and explicit conversions). These are also called weakly typed.

- **Generic collections** – These are the collections that can hold data of same type and we can decide what type of data that collections can hold. These are also called strongly typed.

non-generic collections

The *System.Collections* namespace includes the interfaces and classes for the non-generic collections.

The following diagram illustrates the hierarchy of the interfaces and classes for the non-generic collections.



non-generic collections

- IEnumerator: The IEnumerator interface supports a simple iteration over a non-generic collection. It includes methods and property which can be implemented to support easy iteration using foreach loop.

- IEnumerable: The IEnumerable interface includes GetEnumerator() method which returns an object of IEnumerator.

- ICollection: The ICollection interface is the base interface for all the collections that defines sizes, enumerators, and synchronization methods for all non-generic collections. The Queue and Stack collection implement ICollection inferface.

- IList: The IList interface includes properties and methods to add, insert, remove elements in the collection and also individual element can be accessed by index. The ArrayList and BitArray collections implement IList interface.

- IDictionary: The IDictionary interface represents a non-generic collection of key/value pairs. The Hashtable and SortedList implement IDictionary interface and so they store key/value pairs.

The System.Collections Namespace

Table 9-1. Useful Types of System.Collections

System.Collections Class	Meaning in Life	Key Implemented Interfaces
`ArrayList`	Represents a dynamically sized collection of objects listed in sequential order.	`IList`, `ICollection`, `IEnumerable`, and `ICloneable`
`BitArray`	Manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).	`ICollection`, `IEnumerable`, and `ICloneable`
`Hashtable`	Represents a collection of key/value pairs that are organized based on the hash code of the key.	`IDictionary`, `ICollection`, `IEnumerable`, and `ICloneable`
`Queue`	Represents a standard first-in, first-out (FIFO) collection of objects.	`ICollection`, `IEnumerable`, and `ICloneable`
`SortedList`	Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.	`IDictionary`, `ICollection`, `IEnumerable`, and `ICloneable`
`Stack`	A last-in, first-out (LIFO) stack providing push and pop (and peek) functionality.	`ICollection`, `IEnumerable`, and `ICloneable`

ArrayList Class

- It represents an 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.

Property	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 has a read-only.
Item	Gets or Sets the element at the specified index.

ArrayList Methods

public virtual int Add(object value);
- Adds an object to the end of the ArrayList.

public virtual void AddRange(ICollection c);
- Adds the elements of an ICollection to the end of the ArrayList.

public virtual void Clear(); Removes all elements from the ArrayList.

public virtual bool Contains(object item);
- Determines whether an element is in the ArrayList.

public virtual ArrayList GetRange(int index, int count);
- Returns an ArrayList which represents a subset of the elements in the source ArrayList.

public virtual int IndexOf(object);
- Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it.

public virtual void Insert(int index, object value);
- Inserts an element into the ArrayList at the specified index.

ArrayList Methods

public virtual void InsertRange(int index, ICollection c);
- Inserts the elements of a collection into the ArrayList at the specified index.

public virtual void Remove(object obj);
- Removes the first occurrence of a specific object from the ArrayList.

public virtual void RemoveAt(int index);
- Removes the element at the specified index of the ArrayList.

public virtual void RemoveRange(int index, int count);
- Removes a range of elements from the ArrayList.

public virtual void Reverse();
- Reverses the order of the elements in the ArrayList.

public virtual void SetRange(int index, ICollection c);
- Copies the elements of a collection over a range of elements in the ArrayList.

public virtual void Sort(); Sorts the elements in the ArrayList.

public virtual void TrimToSize();
- Sets the capacity to the actual number of elements in the ArrayList.

Example

```
using System;
using System.Collections;

namespace CollectionApplication
{
  class Program
  {
    static void Main(string[] args)
    {
   ArrayList al = new ArrayList();
  Console.WriteLine("Adding some numbers:");
       al.Add(45);
       al.Add(78);
       al.Add(33);
       al.Add(56);
       al.Add(12);
       al.Add(23);
       al.Add(9);
Console.WriteLine("Capacity: {0} ",
al.Capacity);
Console.WriteLine("Count: {0}", al.Count);
 Console.Write("Content: ");
 foreach (int i in al)
 {
     Console.Write(i + " ");
 }
 Console.WriteLine();
 Console.Write("Sorted Content: ");
  al.Sort();
 foreach (int i in al)
 {
     Console.Write(i + " ");
 }
 Console.WriteLine();
Console.ReadKey();
   }  }  }
```

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Content: 9 12 23 33 45 56 78
```

Hashtable Class

- The Hashtable class represents a collection of key-and-value pairs that are organized based on the hash code of the key.
- It uses the 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.

Property	Description
Count	Gets the number of key-and-value pairs contained in the Hashtable.
isFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable has a read-only.
Item	Gets or Sets the value associated with the specified key.
Keys	Gets an ICollection containing the keys in the Hashtable.
values	Gets an ICollection containing the vlues in the Hashtable.

Hashtable Methods

public virtual void Add(object key, object value);
- Adds an element with the specified key and value into the Hashtable.

public virtual void Clear();
- Removes all elements from the Hashtable.

public virtual bool ContainsKey(object key);
- Determines whether the Hashtable contains a specific key.

public virtual bool ContainsValue(object value);
- Determines whether the Hashtable contains a specific value.

public virtual void Remove(object key);
- Removes the element with the specified key from the Hashtable.

Example

```
using System;
using System.Collections;

namespace CollectionsApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Hashtable ht = new Hashtable();
      ht.Add("001", "Zara Ali");
      ht.Add("002", "Abida Rehman");
      ht.Add("003", "Joe Holzner");
      ht.Add("004", "Mausam Benazir Nur");
      ht.Add("005", "M. Amlan");
      ht.Add("006", "M. Arif");
      ht.Add("007", "Ritesh Saikia");

      if (ht.ContainsValue("Nuha Ali"))
      {
        Console.WriteLine("This student
name is already in the list");
      }
      else
      {          ht.Add("008", "Nuha Ali");
      }
        // Get a collection of the keys.
      ICollection key = ht.Keys;

      foreach (string k in key)
      {
        Console.WriteLine(k + ": " + ht[k]);
      }

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

```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
004: Mausam Benazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

SortedList Class

- The SortedList class represents a collection of key-and-value pairs that are sorted by the keys and are accessible by key and by index.
- 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.

SortedList Properties

Property	Description
Capacity	Gets or sets the capacity of the SortedList.
Count	Gets the number of elements contained in the SortedList.
isFixedSize	Gets a value indicating whether the SortedList has a fixed size.
IsReadOnly	Gets a value indicating whether the SortedList is read-only.
Item	Gets and sets the value associated with a specific key in the SortedList.
Keys	Gets the keys in the SortedList.
values	Gets the values in the SortedList.

SortedList Methods

public virtual void Add(object key, object value);
- Adds an element with the specified key and value into the SortedList

public virtual void Clear();
- Removes all elements from the SortedList.

public virtual bool ContainsKey(object key);
- Determines whether the SortedList contains a specific key.

public virtual bool ContainsValue(object value);
- Determines whether the SortedList contains a specific value.

public virtual object GetByIndex(int index);
- Gets the value at the specified index of the SortedList.

public virtual object GetKey(int index);
- Gets the key at the specified index of the SortedList.

public virtual IList GetKeyList();
- Gets the keys in the SortedList.

SortedList Methods

public virtual IList GetValueList();
- Gets the values in the SortedList.

public virtual int IndexOfKey(object key);
- Returns the zero-based index of the specified key in the SortedList.

public virtual int IndexOfValue(object value);
- Returns the zero-based index of the first occurrence of the specified value in the SortedList.

public virtual void Remove(object key);
- Removes the element with the specified key from the SortedList.

public virtual void RemoveAt(int index);
- Removes the element at the specified index of SortedList.

public virtual void TrimToSize();
- Sets the capacity to the actual number of elements in the SortedList.

Example

```
using System;
using System.Collections;

namespace CollectionsApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      SortedList sl = new SortedList();

      sl.Add("001", "Zara Ali");
      sl.Add("002", "Abida Rehman");
      sl.Add("003", "Joe Holzner");
      sl.Add("004", "Mausam Benazir Nur");
      sl.Add("005", "M. Amlan");
      sl.Add("006", "M. Arif");
      sl.Add("007", "Ritesh Saikia");

      if (sl.ContainsValue("Nuha Ali"))
      {
        Console.WriteLine("This student
name is already in the list");
      }
      else
      {
        sl.Add("008", "Nuha Ali");
      }

      // get a collection of the keys.
      ICollection key = sl.Keys;

      foreach (string k in key)
      {
        Console.WriteLine(k + ": " + sl[k]);
      }    }  } }
```

```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
004: Mausam Benazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

Stack Class

- 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.

Property:

Count Gets the number of elements contained in the Stack.

Stack Methods

public virtual void Clear();
- Removes all elements from the Stack.

public virtual bool Contains(object obj);
- Determines whether an element is in the Stack.

public virtual object Peek();
- Returns the object at the top of the Stack without removing it.

public virtual object Pop();
- Removes and returns the object at the top of the Stack.

public virtual void Push(object obj);
- Inserts an object at the top of the Stack.

public virtual object[] ToArray();
- Copies the Stack to a new array.

Example

```
using System;
using System.Collections;

namespace CollectionsApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Stack st = new Stack();

      st.Push('A');
      st.Push('M');
      st.Push('G');
      st.Push('W');

      Console.WriteLine("Current stack: ");
      foreach (char c in st)
      {
        Console.Write(c + " ");
      }

      Console.WriteLine();
      st.Push('V');
      st.Push('H');
      Console.WriteLine("The next poppable value in
stack: {0}", st.Peek());
      Console.WriteLine("Current stack: ");
      foreach (char c in st)
      {
        Console.Write(c + " ");
      }
      Console.WriteLine();
       Console.WriteLine("Removing values ");
      st.Pop();
      st.Pop();
      st.Pop();

      Console.WriteLine("Current stack: ");
      foreach (char c in st)
      {
        Console.Write(c + " ");
      }    }  }  }
```

```
Current stack:
W G M A
The next poppable value in stack: H
Current stack:
H V W G M A
Removing values
Current stack:
G M A
```

Queue Class

- 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.

Property

Count Gets the number of elements contained in the Queue.

Queue Methods

public virtual void Clear();
- Removes all elements from the Queue.

public virtual bool Contains(object obj);
- Determines whether an element is in the Queue.

public virtual object Dequeue();
- Removes and returns the object at the beginning of the Queue.

public virtual void Enqueue(object obj);
- Adds an object to the end of the Queue.

public virtual object[] ToArray();
- Copies the Queue to a new array.

public virtual void TrimToSize();
- Sets the capacity to the actual number of elements in the Queue.

Example

```
using System;
using System.Collections;

namespace CollectionsApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Queue q = new Queue();

      q.Enqueue('A');
      q.Enqueue('M');
      q.Enqueue('G');
      q.Enqueue('W');

      Console.WriteLine("Current queue: ");
      foreach (char c in q)
     Console.Write(c + " ");
    Console.WriteLine();
```

```
q.Enqueue('V');
q.Enqueue('H');
Console.WriteLine("Current queue: ");
 foreach (char c in q)
Console.Write(c + " ");
 Console.WriteLine();
Console.WriteLine("Removing some values
");
char ch = (char)q.Dequeue();
Console.WriteLine("The removed value:
{0}", ch);
ch = (char)q.Dequeue();
Console.WriteLine("The removed value:
{0}", ch);

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

Current queue:
A M G W

Current queue:

A M G W V H

Removing values

The removed value: A

The removed value: M

BitArray Class

- The BitArray class manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (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.

Property	Description
Count	Gets the number of elements contained in the BitArray.
IsReadOnly	Gets a value indicating whether the BitArray is read-only.
Item	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.

BitArray Methods

public BitArray And(BitArray value);
- Performs the bitwise AND operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.

public bool Get(int index);
- Gets the value of the bit at a specific position in the BitArray.

public BitArray 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.

public BitArray Or(BitArray value);
- Performs the bitwise OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.

public void Set(int index, bool value);
- Sets the bit at a specific position in the BitArray to the specified value.

public void SetAll(bool value);
- Sets all bits in the BitArray to the specified value.

public BitArray Xor(BitArray value);
- Performs the bitwise eXclusive OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.

Example

```
using System;
using System.Collections;

namespace CollectionsApplication
{   class Program
  {      static void Main(string[] args)
    {         //creating two  bit arrays of size 8
      BitArray ba1 = new BitArray(8);
      BitArray ba2 = new BitArray(8);
      byte[] a = { 60 };
      byte[] b = { 13 };
 //storing the values 60, and 13 into the bit arrays
      ba1 = new BitArray(a);
      ba2 = new BitArray(b);
      //content of ba1
      Console.WriteLine("Bit array ba1: 60");
      for (int i = 0; i < ba1.Count; i++)
      {          Console.Write("{0, -6} ", ba1[i]);        }
      Console.WriteLine();
      //content of ba2
      Console.WriteLine("Bit array ba2: 13");
      for (int i = 0; i < ba2.Count; i++)
      {
        Console.Write("{0, -6} ", ba2[i]);
}
Console.WriteLine();
BitArray ba3 = new BitArray(8);
ba3 = ba1.And(ba2);
 //content of ba3
Console.WriteLine("Bit array ba3 after AND operation:
12");
 for (int i = 0; i < ba3.Count; i++)
 {          Console.Write("{0, -6} ", ba3[i]);        }
Console.WriteLine();
ba3 = ba1.Or(ba2);
//content of ba3
Console.WriteLine("Bit array ba3 after OR operation:
61");
for (int i = 0; i < ba3.Count; i++)
 {          Console.Write("{0, -6} ", ba3[i]);        }
 Console.WriteLine();
 Console.ReadKey();
      }   }   }
```

System.Collections.Generic classes

CLASS NAME	DESCRIPTION
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
SortedList<T>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
Stack<T>	It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
HashSet<T>	It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.

Collections – List

- C# List<T> class is used to store and fetch elements.
- It can have duplicate elements.
- It is found in System.Collections.Generic namespace.
- Let's see an example of generic List<T> class that stores elements using Add() method and iterates the list using for-each loop.

- using Collection initializer.

```
var names = new List<string>() {"Sonoo", "Vimal", "Ratan", "Love" };
```

Collections – List

Property	Usage
Items	Gets or sets the element at the specified index
Count	Returns the total number of elements exists in the List<T>

Method	Usage
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the speciied element exists or not in a List<T>.
Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
InsertRange	Inserts elements of another collection at the specified index.
Remove	Removes the first occurence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match with the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.
TrueForAll	Determines whether every element in theÂ List<T> matches the conditions defined by the specified predicate.

Collections – List

```
using System;
using System.Collections.Generic;
  public class ListExample
{       public static void Main(string[] args)
    {
        var names = new List<string>();  // Create a list of strings
        names.Add("Sonoo Jaiswal");
        names.Add("Ankit");
        names.Add("Peter");
        names.Add("Irfan");
        foreach (var name in names)   // Iterate list element using foreach loop
        {               Console.WriteLine(name);
        }   }   }
```

Output:

```
Sonoo Jaiswal
Ankit
Peter
Irfan
```

Collections – HashSet

- C# HashSet class can be used to store, remove or view elements.
- It does not store duplicate elements.
- It is suggested to use HashSet class if you have to store only unique elements.
- It is found in System.Collections.Generic namespace.
- Let's see an example of generic HashSet<T> class that stores elements using Add() method and iterates elements using for-each loop.

- using Collection initializer.

```
var names = new HashSet<string>(){"Sonoo", "Ankit", "Peter", "Irfan"};
```

Collections – HashSet

Property	Description
Count	Gets the total count of key/value pairs in the Hashtable.
IsReadOnly	Gets boolean value indicating whether the Hashtable is read-only.
Item	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection of keys in the Hashtable.
Values	Gets an ICollection of values in the Hashtable.

Methods	Usage
Add	Adds an item with a key and value into the hashtable.
Remove	Removes the item with the specified key from the hashtable.
Clear	Removes all the items from the hashtable.
Contains	Checks whether the hashtable contains a specific key.
ContainsKey	Checks whether the hashtable contains a specific key.
ContainsValue	Checks whether the hashtable contains a specific value.
GetHash	Returns the hash code for the specified key.

Collections – HashSet

```
using System;
using System.Collections.Generic;
 public class HashSetExample
{       public static void Main(string[] args)
   {
      var names = new HashSet<string>();  // Create a set of strings
       names.Add("Sonoo");
       names.Add("Ankit");
       names.Add("Peter");
       names.Add("Irfan");
       names.Add("Ankit");//will not be added
       foreach (var name in names)   // Iterate HashSet elements using foreach loop
       {
          Console.WriteLine(name);
       }     }  }
```

Output:

```
Sonoo
Ankit
Peter
Irfan
```

Collections – Stack

- # Stack<T> class is used to push and pop elements.
- It uses the concept of Stack that arranges elements in LIFO (Last In First Out) order.
- It can have duplicate elements.
- It is found in System.Collections.Generic namespace.

- Let's see an example of generic Stack<T> class that stores elements using Push() method, removes elements using Pop() method and iterates elements using for-each loop.

Collections – Stack

Property	Usage
Count	Returns the total count of elements in the Stack.

Method	Usage
Push	Inserts an item at the top of the stack.
Peek	Returns the top item from the stack.
Pop	Removes and returns items from the top of the stack.
Contains	Checks whether an item exists in the stack or not.
Clear	Removes all items from the stack.

Collections – Stack

```
using System;
using System.Collections.Generic;
public class StackExample
{
   public static void Main(string[] args)
   {
      Stack<string> names = new Stack<string>();
      names.Push("Sonoo");
      names.Push("Peter");
      names.Push("James");
      names.Push("Ratan");
      names.Push("Irfan");
     foreach (string name in names)
      {
         Console.WriteLine(name);
      }
      Console.WriteLine("Peek element: "+names.Peek());
      Console.WriteLine("Pop: "+ names.Pop());
      Console.WriteLine("After Pop, Peek element: " + names.Peek());
   } }
```

Output:
Sonoo
Peter
James
Ratan
Irfan
Peek element: Irfan
Pop: Irfan
After Pop,
Peek element: Ratan

Collections – Queue

- C# Queue<T> class is used to Enqueue and Dequeue elements.
- It uses the concept of Queue that arranges elements in FIFO (First In First Out) order.
- It can have duplicate elements.
- It is found in System.Collections.Generic namespace.

- Let's see an example of generic Queue<T> class that stores elements using Enqueue() method, removes elements using Dequeue() method and iterates elements using for-each loop.

Collections – Queue

Property	Usage
Count	Returns the total count of elements in the Queue.

Method	Usage
Enqueue	Adds an item into the queue.
Dequeue	Removes and returns an item from the beginning of the queue.
Peek	Returns an first item from the queue
Contains	Checks whether an item is in the queue or not
Clear	Removes all the items from the queue.
TrimToSize	Sets the capacity of the queue to the actual number of items in the queue.

Collections – Queue

```
using System;
using System.Collections.Generic;
 public class QueueExample
{
   public static void Main(string[] args)
   {
      Queue<string> names = new Queue<string>();
      names.Enqueue("Sonoo");
      names.Enqueue("Peter");
      names.Enqueue("James");
      names.Enqueue("Ratan");
      names.Enqueue("Irfan");
      foreach (string name in names)
      {
         Console.WriteLine(name);
      }
     Console.WriteLine("Peek element: "+names.Peek());
     Console.WriteLine("Dequeue: "+ names.Dequeue());
      Console.WriteLine("After Dequeue, Peek element: " + names.Peek());
   } }
```

Output:
Sonoo
Peter
James
Ratan
Irfan
Peek element: Sonoo
Dequeue: Sonoo
After Dequeue,
Peek element: Peter

Collections – LinkedList

- C# LinkedList<T> class uses the concept of linked list.
- It allows us to insert and delete elements fastly.
- It can have duplicate elements.
- It is found in System.Collections.Generic namespace.
- It allows us to add and remove element at before or last index.
- Let's see an example of generic LinkedList<T> class that stores elements using AddLast() and AddFirst() methods and iterates elements

Collections – LinkedList

using for-each loop.

```
using System;
using System.Collections.Generic;
public class LinkedListExample
{
   public static void Main(string[] args)
   {
      var names = new LinkedList<string>();  // Create a list of strings
      names.AddLast("Sonoo Jaiswal");
      names.AddLast("Ankit");
      names.AddLast("Peter");
      names.AddLast("Irfan");
      names.AddFirst("John");//added to first index
     foreach (var name in names)   // Iterate list element using foreach loop
      {
         Console.WriteLine(name);
      }     }  }
```

Output:

```
John
Sonoo Jaiswal
Ankit
Peter
Irfan
```

Collections – SortedSet

- C# SortedSet class can be used to store, remove or view elements.
- It maintains ascending order and does not store duplicate elements.
- It is suggested to use SortedSet class if you have to store unique elements and maintain ascending order.
- It is found in System.Collections.Generic namespace.
- Let's see an example of generic SortedSet<T> class that stores elements using Add() method and iterates elements using for-each loop.

Collections – SortedSet

Property	Description
Capacity	Gets or sets the number of elements that the SortedList instance can store.
Count	Gets the number of elements actually contained in the SortedList.
IsFixedSize	Gets a value indicating whether the SortedList has a fixed size.
IsReadOnly	Gets a value indicating whether the SortedList is read-only.
Item	Gets or sets the element at the specified key in the SortedList.
Keys	Get list of keys of SortedList.
Values	Get list of values in SortedList.

Collections – SortedSet

Method	Description
`void Add(object key, object value)`	Add key-value pairs into SortedList.
`void Remove(object key)`	Removes element with the specified key.
`void RemoveAt(int index)`	Removes element at the specified index.
`bool Contains(object key)`	Checks whether specified key exists in SortedList.
`void Clear()`	Removes all the elements from SortedList.
`object GetByIndex(int index)`	Returns the value by index stored in internal array
`object GetKey(int index)`	Returns the key stored at specified index in internal array
`int IndexOfKey(object key)`	Returns an index of specified key stored in internal array
`int IndexOfValue(object value)`	Returns an index of specified value stored in internal array

Collections – SortedSet

```
using System;
using System.Collections.Generic;
public class SortedSetExample
{
   public static void Main(string[] args)
   {
      // Create a set of strings
      var names = new SortedSet<string>();
      names.Add("Sonoo");
      names.Add("Ankit");
      names.Add("Peter");
      names.Add("Irfan");
      names.Add("Ankit");//will not be added
       // Iterate SortedSet elements using foreach loop
      foreach (var name in names)
      {
         Console.WriteLine(name);
      }    } }
```

Output:

```
Ankit
Irfan
Peter
Sonoo
```

Indexers

- Indexer Concept is object act as an array.
- Indexer an object to be indexed in the same way as an array.
- Indexer modifier can be private, public, protected or internal.
- The return type can be any valid C# types.
- Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.
- It is same as property except that it defined with **this** keyword with square bracket and parameters.

Indexers

Syntax:

```
Public <return type> this[<parameter type> index]
{
   Get{
      // return the value from the specified index
   }
   Set{
      // set values at the specified index
   }
}
```

Indexers

```
class StringDataStore{
private string[] strArr = new string[10];
internal data storage
public StringDataStore()    {     }
public string this[int index]
{
 get
{
 if (index < 0 &&  index >= strArr.Length)
throw new
IndexOutOfRangeException("Cannot store
more than 10 objects");
return strArr[index];
}
set
{
if (index < 0 &&  index >= strArr.Length)
throw new
IndexOutOfRangeException("Cannot store
more than 10 objects");
strArr[index] = value;
}}}
class Program
{
static void Main(string[] args)
{
StringDataStore strStore = new
StringDataStore();
strStore[0] = "One";
strStore[1] = "Two";
strStore[2] = "Three";
strStore[3] = "Four";
for(int i = 0; i < 10 ; i++)
Console.WriteLine(strStore[i]);    }}
```

Delegates

- Delegate is a *reference to the method*.
- It works like *function pointer* in C and C++. But it is objected-oriented, secured and type-safe than function pointer.
- For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both.
- The best use of delegate is to use as event.
- Internally a delegate declaration defines a class which is the derived class of **System.Delegate**.

Syntax:

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

Delegates

```
using System;
delegate int Calculator(int n);
public class DelegateExample
{
 static int number = 100;
 public static int add(int n)
 {
 number = number + n;
 return number;
 }
 public static int mul(int n)
 {
 number = number * n;
 return number;
 }
 public static int getNumber()
 {
 return number;
 }
 public static void Main(string[] args)

 Calculator c1 = new Calculator(add);
 Calculator c2 = new Calculator(mul);
 c1(20);//calling method using delegate
 Console.WriteLine("After c1 delegate, N
 umber is: " + getNumber());
 c2(3);
 Console.WriteLine("After c2 delegate, N
 umber is: " + getNumber());
     }
 }
```

Delegates

```
using System;
namespace Delegates
{
// Delegate Definition
public delegate int operation(int x, int y);
class Program
{
// Method that is passes as an Argument
// It has same signature as Delegates
static int Addition(int a, int b)
{
return a + b;
}
static void Main(string[] args)
{
// Delegate instantiation
operation obj = new operation(Program.Addition);
// output
Console.WriteLine("Addition is={0}",obj(23,27));
Console.ReadLine();
}
}
}
```

MULTICAST DELEGATE

- **Multicast Delegate** is an extension of normal delegates. It combines more than one method at a single moment of time.

IMPORTANT FACT ABOUT MULTICAST DELEGATE

- In Multicasting, Delegates can be combined and when you call a delegate, a whole list of methods is called.
- All methods are called in FIFO (First in First Out) order.
- + or += Operator is used for adding methods to delegates.
- – or -= Operator is used for removing methods from the delegates list.

MULTICAST DELEGATE

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

namespace Multicast_Delegates
{
class TestDeleGate
{
public delegate void ShowMessage(string s);
public void message1(string msg)
{
Console.WriteLine("1st Message is : {0}", msg);
}
public void message2(string msg)
{
Console.WriteLine("2nd Message is : {0}", msg);
}
public void message3(string msg)
{
Console.WriteLine("3rd Message is : {0}", msg);
}  }
class Program
{
static void Main(string[] args)
{
TestDeleGate td = new TestDeleGate();
TestDeleGate.ShowMessage message = null;
message += new
TestDeleGate.ShowMessage(td.message1);
message += new
TestDeleGate.ShowMessage(td.message2);
message += new
TestDeleGate.ShowMessage(td.message3);
message("Hello Multicast Delegates");
message -= new
TestDeleGate.ShowMessage(td.message2);
Console.WriteLine("----------------------");
message("Message 2 Removed");
Console.ReadKey();
}  }  }
```

MULTICAST DELEGATE

```
namespace MulticastDelegateDemo
{
public delegate void MathDelegate(int No1, int No2);
public class Program
{
public static void Add(int x, int y)
{
Console.WriteLine("THE SUM IS : " + (x + y));
}
public static void Sub(int x, int y)
{
Console.WriteLine("THE SUB IS : " + (x - y));
}
public void Mul(int x, int y)
{
Console.WriteLine("THE MUL IS : " + (x * y));
}
public void Div(int x, int y)
{
Console.WriteLine("THE DIV IS : " + (x / y));
}
static void Main(string[] args)
{
Program p = new Program();
MathDelegate del1 = new MathDelegate(Add);
MathDelegate del2 = new MathDelegate(Program.Sub);
MathDelegate del3 = p.Mul;
MathDelegate del4 = new MathDelegate(p.Div); ;
//In this example del5 is a multicast delegate. We can
use +(plus)
// operator to chain delegates together and -(minus)
operator to remove.
MathDelegate del5 = del1 + del2 + del3 + del4;
del5.Invoke(20, 5);
Console.WriteLine();
del5 -= del2;
del5(22, 7);
Console.ReadKey();
} } }
```

Anonymous method in Delegate

- An anonymous method is a method which doesn't contain any name which is introduced in C# 2.0.
- It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.
- An Anonymous method is defined using the delegate keyword and the user can assign this method to a variable of the delegate type.
- Using this method you can create a delegate object without writing separate methods.

Syntax:

```
delegate(parameter_list){
   // Code..
};
```

Anonymous method in Delegate

```
public delegate void Print(int value);

static void Main(string[] args)
{
    int i = 10;

    Print prnt = delegate(int val)
{
        val += i;
        Console.WriteLine("Anonymous method: {0}", val);
    };

    prnt(100);
}
```

Events

- Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications.
- Applications need to respond to events when they occur.
- For example, interrupts. Events are used for inter-process communication.

Using Delegates with Events

- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the *publisher class*.
- Some other class that accepts this event is called the *subscriber class*.
- Events use the publisher-subscriber model.

Events

- A publisher is an object that contains the definition of the event and the delegate.
- The event-delegate association is also defined in this object.
- A publisher class object invokes the event and it is notified to other objects.
- A subscriber is an object that accepts the event and provides an event handler.
- The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

- To declare an event inside a class, first a delegate type for the event must be declared. For example,

```
public delegate string MyDel(string str);
```

- Next, the event itself is declared, using the event keyword −

```
event MyDel MyEvent;
```

Events

```
using System;
namespace SampleApp
 {
  public delegate string MyDel(string str);
  class EventProgram
 {
    event MyDel MyEvent;
    public EventProgram()
{
      this.MyEvent += new MyDel(this.WelcomeUser);
    }
    public string WelcomeUser(string username)
 {
      return "Welcome to " + username;
    }
    static void Main(string[] args)
{
      EventProgram obj1 = new EventProgram();
      string result = obj1.MyEvent("C# & .NET Technologies");
      Console.WriteLine(result);
    }   } }
```

Lambda Expressions

- The **Lambda Expression in C#** is the shorthand for writing the anonymous function.
- So we can say that the Lambda Expression in C# is nothing but to simplify the anonymous function in C#.
- To create a lambda expression in C#, we need to specify the input parameters (if any) on the left side of the lambda operator =>, and we need to put the expression or statement block on the other side.

Delegate

```
namespace LambdaExpressionDemo
{
public class LambdaExpression
{
public delegate string GreetingsDelegate(string name);
static void Main(string[] args)
{
GreetingsDelegate obj = new
GreetingsDelegate(LambdaExpression.Greetings);
string GreetingsMessage = obj.Invoke("Pranaya");
Console.WriteLine(GreetingsMessage);
Console.ReadKey();
}
public static string Greetings(string name)
{
return "Hello @" + name + " welcome to Dotnet Tutorials";
} }  }
```

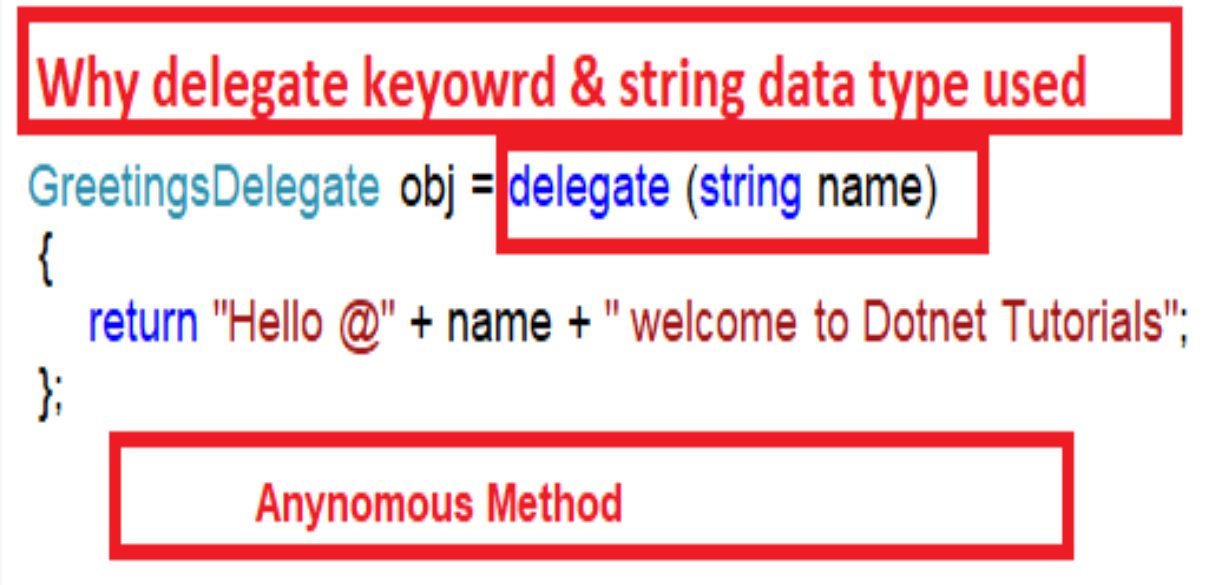
```
Hello @Pranaya Welcome to Dotnet Tutorials
```

Anonymous method in Delegate

```
namespace LambdaExpressionDemo
{
public class LambdaExpression
{
public delegate string GreetingsDelegate(string name);
static void Main(string[] args)
{
GreetingsDelegate obj = delegate (string name)
{
return "Hello @" + name + " welcome to Dotnet Tutorials";
};
string GreetingsMessage = obj.Invoke("Pranaya");
Console.WriteLine(GreetingsMessage);
Console.ReadKey();
}          }          }
```

```
Hello @Pranaya Welcome to Dotnet Tutorials
```

Anonymous method in Delegate



Lambda expression

```
namespace LambdaExpressionDemo
{
public class LambdaExpression
{
public delegate string GreetingsDelegate(string name);
static void Main(string[] args)
{
GreetingsDelegate obj = (name) =>
{
return "Hello @" + name + " welcome to Dotnet Tutorials";
};
string GreetingsMessage = obj.Invoke("Pranaya");
Console.WriteLine(GreetingsMessage);
Console.ReadKey();
}
public static string Greetings(string name)
{
return "Hello @" + name + " welcome to Dotnet Tutorials";
}           }           }
```

```
Hello @Pranaya Welcome to Dotnet Tutorials
```

Lambda expression