VT Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming

Collections and Generics

Session 8

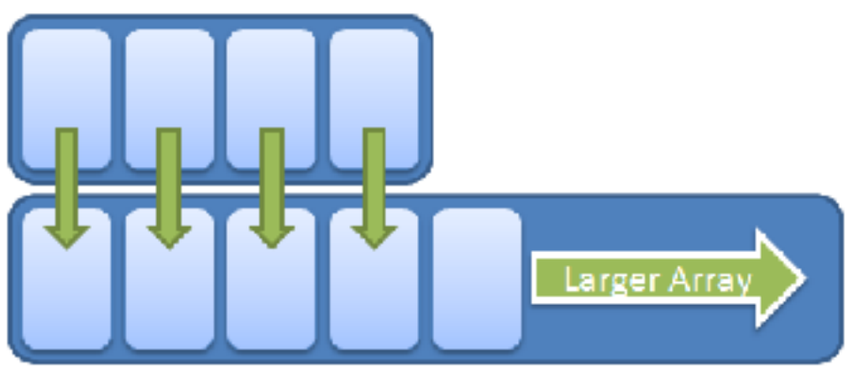
Objectives

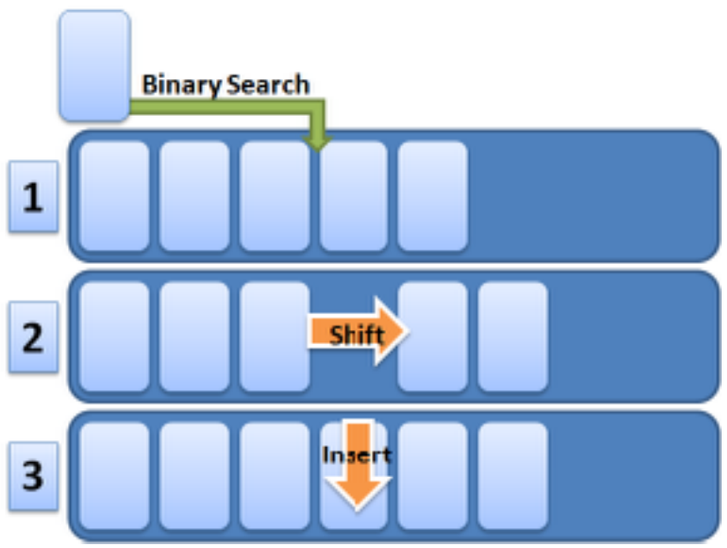
VTC Academy
Thinking Ahead, Beyond Boundaries

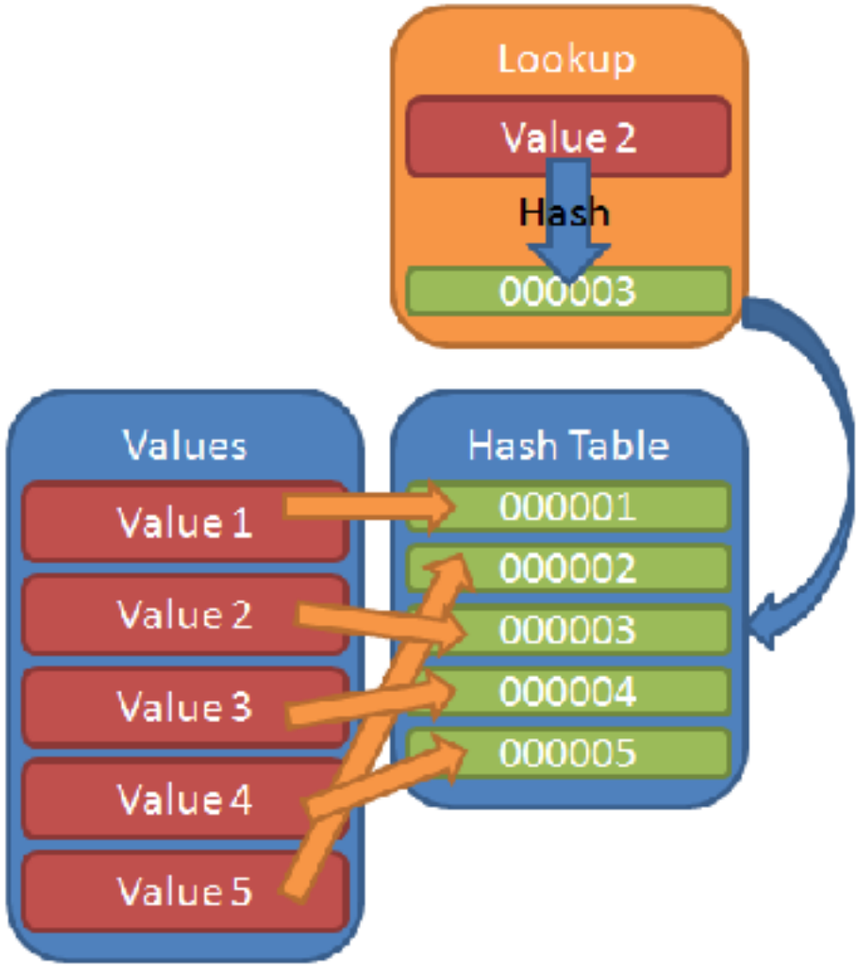
- Collections
- Generics
- Iterators

Collections

- Collection classes are specialized classes for data storage and retrieval
- 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







Collections

Arrays	Collections
Cannot be resized at runtime	Can be resized at runtime
Individual elements are of the same data type	Individual elements can be of different data types
Do not contain any methods for operations on elements	Contain methods for operations on elements

Collections

- Kinds of collectons:
  - System.Collections Classes
  - System.Collections.Generic Classes
  - System.Collections.Concurrent Classes
- Most common collection classes:
  - ArrayList
  - HashTable
  - SortedList
  - Stack
  - Queue

ArrayList

- ArrayList represents an ordered collection of an object that can be indexed individually.
- ArrayList 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.
- ArrayList also allows dynamic memory allocation, adding, searching and sorting items in the list
- Syntax to create ArrayList:

```
ArrayList alist = new ArrayList();
```

Properties of ArrayList Class

VTC Academy
Thinking Ahead, Beyond Boundaries

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 is read-only
Item	Gets or sets the element at the specified index

Common Methods of ArrayList Class

VTC Academy
Thinking Ahead, Beyond Boundaries

No.	Methods
1	**public virtual int Add(object value)** Adds an object to the end of the ArrayList
3	**public virtual void Clear()** Removes all elements from the ArrayList
4	**public virtual bool Contains(object item)** Determines whether an element is in the ArrayList
5	**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
6	**public virtual void Insert(int index, object value)** Inserts an element into the ArrayList at the specified index
7	**public virtual void Remove(object obj)** Removes the first occurrence of a specific object from the ArrayList
8	**public virtual void RemoveAt(int index)** Removes the element at the specified index of the ArrayList
9	**public virtual void Reverse()** Reverses the order of the elements in the ArrayList
10	**public virtual void Sort()** Sorts the elements in the ArrayList

ArrayList Example

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

HashTable

- 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**, eand 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.

- Syntax to create HashTable:

```
Hashtable htable = new Hashtable();
```

Properties of the Hashtable Class

VTC Academy
Thinking Ahead, Beyond Boundaries

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 is 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 values in the Hashtable

Methods of the Hashtable Class

No.	Method
1	**public virtual void Add(object key, object value)** Adds an element with the specified key and value into the Hashtable
2	**public virtual void Clear()** Removes all elements from the Hashtable
3	**public virtual bool ContainsKey(object key)** Determines whether the Hashtable contains a specific key
4	**public virtual bool ContainsValue(object value)** Determines whether the Hashtable contains a specific value
5	**public virtual void Remove(object key)** Removes the element with the specified key from the Hashtable

HashTable Example

```
using System;
using System.Collections;
namespace HashTableDemo {
   class Program {
      static void Main(string[] args)
      {
         Hashtable htable = new Hashtable();
         htable.Add("001", "Zara Ali");
         htable.Add("002", "Abida Rehman");
         htable.Add("003", "Bill Gates");
         htable.Add("004", "Mausam Benazir Nur");
         htable.Add("005", "Ritesh Saikia");
         if (htable.ContainsValue("Bill Gates"))
         {
            Console.WriteLine("This student name is already in the list”);
         }
         else
         {
            htable.Add("006", "Bill Gates");
         }
         // Get a collection of the keys.
         ICollection key = htable.Keys;
         foreach (string k in key)
         {
            Console.WriteLine(k + ": " + htable[k]);
         }
         Console.ReadKey();
      }
   }
}
```

SortedList

- 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.
- Syntax to create HashTable:

```
SortedList sl = new SortedList();
```

Properties of the SortedList Class

VTC Academy
Thinking Ahead, Beyond Boundaries

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

Methods of the SortedList Class

VTC Academy
Thinking Ahead, Beyond Boundaries

No.	Methods
1	**public virtual void Add(object key, object value)** Adds an element with the specified key and value into the SortedList
2	**public virtual void Clear()** Removes all elements from the SortedList
3	**public virtual bool ContainsKey(object key)** Determines whether the SortedList contains a specific key
4	**public virtual bool ContainsValue(object value)** Determines whether the SortedList contains a specific value
5	**public virtual object GetKey(int index)** Gets the key at the specified index of the SortedList
7	**public virtual IList GetKeyList()** Gets the keys in the SortedList
8	**public virtual IList GetValueList()** Gets the values in the SortedList
9	**public virtual void Remove(object key)** Removes the element with the specified key from the SortedList

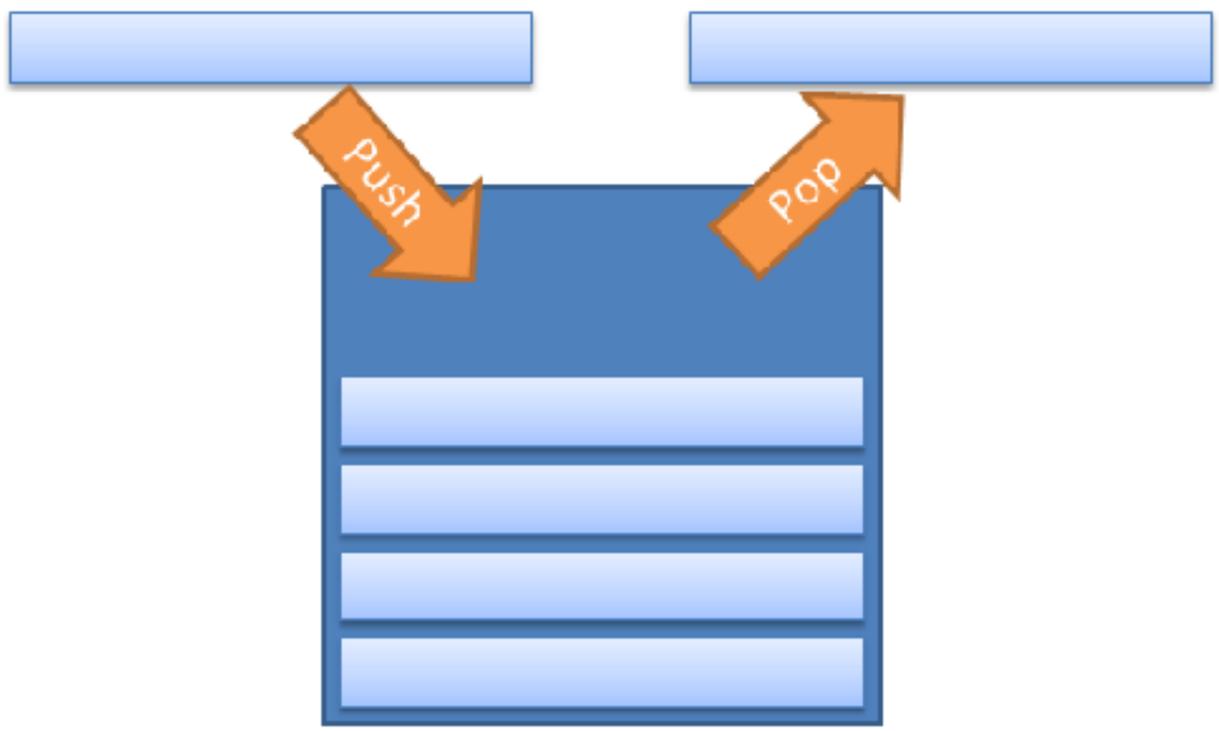
SortedList Example


```
using System;
using System.Collections;
namespace SortedListDemo {
   class Program {
      static void Main(string[] args)
      {
         SortedList slist = new SortedList();
         slist.Add("001", "Zara Ali");
         slist.Add("002", "Abida Rehman");
         slist.Add("003", "Bill Gates");
         slist.Add("004", "Mausam Benazir Nur");
         slist.Add("005", "Ritesh Saikia");
         if (slist.ContainsValue("Bill Gates"))
         {
            Console.WriteLine("This student name is already in the list");
         }
         else
         {
            slist.Add("006", "Bill Gates");
         }
         // get a collection of the keys
         ICollection key = slist.Keys;
         foreach (string k in key)
         {
            Console.WriteLine(k + ": " + slist[k]);
         }
      }
   }
}
```

Stack

- Stack represents a last-in, first out (LIFO) collection of object
- Stack 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.
- Syntax to create Stack:

```
Stack stack = new Stack();
```



Methods of the Stack Class

VTC Academy
Thinking Ahead, Beyond Boundaries

No.	Methods
1	**public virtual void Clear()** Removes all elements from the Stack
2	**public virtual bool Contains(object obj)** Determines whether an element is in the Stack
3	**public virtual object Peek()** Returns the object at the top of the Stack without removing it
4	**public virtual object Pop()** Removes and returns the object at the top of the Stack
5	**public virtual void Push(object obj)** Inserts an object at the top of the Stack
6	**public virtual object[] ToArray()** Copies the Stack to a new array

SortedList Example

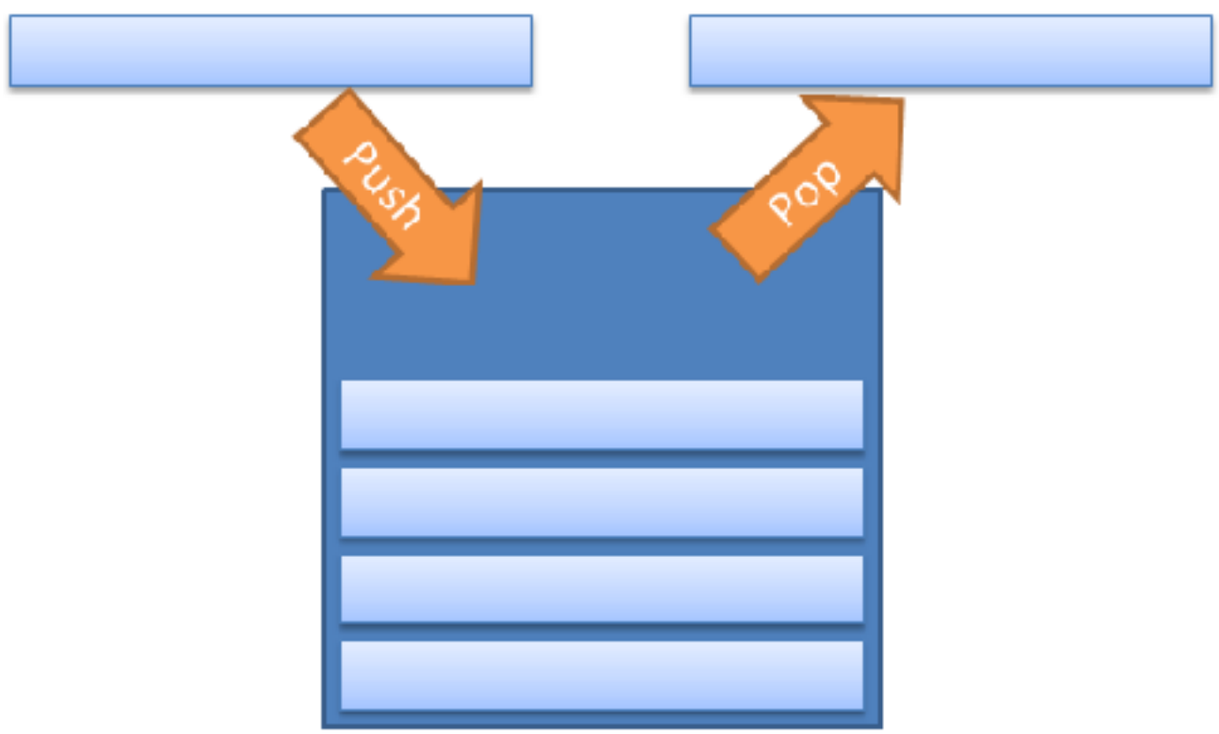
VTC Academy
Thinking Ahead, Beyond Boundaries

```
using System;
using System.Collections;
namespace StackDemo {
   class Program {
      static void Main(string[] args) {
         Stack stack = new Stack();
         stack.Push('A');
         stack.Push('M');
         stack.Push('G');
         stack.Push('W');
         Console.WriteLine("Current stack: ");
         foreach (char c in stack) {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         stack.Push('V');
         stack.Push('H');
         Console.WriteLine("The next poppable value in stack: {0}", stack.Peek());
         Console.WriteLine("Current stack: ");
         foreach (char c in stack) {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         Console.WriteLine("Removing values ");
         stack.Pop();
         stack.Pop();
         stack.Pop();
         Console.WriteLine("Current stack: ");
         foreach (char c in stack) {
            Console.Write(c + " ");
         }
      }
   }
}
```

Queue

- Queue 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**.
- Syntax to create Queue:

```
Queue queue = new Queue();
```



Methods of the Queue Class

VTC Academy
Thinking Ahead, Beyond Boundaries

Sr.No.	Methods
1	**public virtual void Clear()** Removes all elements from the Queue
2	**public virtual bool Contains(object obj)** Determines whether an element is in the Queue
3	**public virtual object Dequeue()** Removes and returns the object at the beginning of the Queue.
4	**public virtual void Enqueue(object obj)** Adds an object to the end of the Queue
5	**public virtual object[] ToArray()** Copies the Queue to a new array
6	**public virtual void TrimToSize()** Sets the capacity to the actual number of elements in the Queue

Queue Example

```
using System;
using System.Collections;
namespace QueueDemo {
   class Program {
      static void Main(string[] args) {
         Queue queue = new Queue();
         queue.Enqueue('A');
         queue.Enqueue('M');
         queue.Enqueue('G');
         queue.Enqueue('W');
         Console.WriteLine("Current queue: ");
         foreach (char c in queue) {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         queue.Enqueue('V');
         queue.Enqueue('H');
         Console.WriteLine("Current queue: “);
         foreach (char c in queue) {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         Console.WriteLine("Removing some values ");
         char ch = (char) queue.Dequeue();
         Console.WriteLine("The removed value: {0}", ch);
         ch = (char) queue.Dequeue();
         Console.WriteLine("The removed value: {0}", ch);
         Console.ReadKey();
      }
   }
}
```

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

- Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.

- For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

Generics Example

VTC Academy
Thinking Ahead, Beyond Boundaries

```
// Declare the generic class.
public class GenericList<T>
{
    void Add(T input) {…}
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
    }
}
```

Generic Classes

- System.Collection.Generic namespace consists of classes and interfaces that allow you to define customized generic collections
- Class:
  - List<T>, LinkedList<T>
  - Stack<T>, Queue<T>
  - HashSet<T>, SortedSet<T>
  - Comparer<T>, Dictionary<TKey,TValue>
- Interface:
  - ICollection<T>, IList<T>
  - IEnumerable<T>, IComparer<T>
  - Dictionary<TKey,TValue>

Creating Generic Types

- A generic declaration always accepts a type parameter, which is a placeholder for the required data type
- Process of creating a generic type begins with a generic type definition containing type parameters
- Syntax to create generic types:

```
<Class Name, Method Name or Interace><<T>>
```

- where:
  - T is the type of a method parameter in the generic class
  - T is the type which generic class will use
- Example:

```
List<int> genList = new List<int>();
or
MyGenericClass<string> genClass = new MyGenericClass<string>();
```

Benefits of Generics

- No casting and Boxing
- Increased Performance
- Strongly-Typed Model
- Less Run-time errors

Benefits of Generics

- Use ArrayList collection:

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
// Add an integer to the list
list.Add(3);
// Add a string to the list. This will compile, but may cause an error later
list.Add("It is raining in Redmond.");
int t = 0;
// This causes an InvalidCastException to be returned
foreach (int x in list)
{
    t += x;
}
```

- Use generic list collection:

```
List<int> list1 = new List<int>();
// No boxing, no casting
list1.Add(3);
// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

Constraints on Type Parameters

Constraint	Description
where T: struct	Type argument must be a value type. Any value type except Nullable can be specified.
where T : class	Type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	Type argument must have a public parameterless constructor. When used together with other constraints, new() constraint must be specified last.
where T : <base class name>	Type argument must be or derive from the specified base class.
where T : <interface name>	Type argument must be or implement the specified interface. Multiple interface constraints can be specified.
where T : U	Type argument supplied for T must be or derive from the argument supplied for U.

Generic Methods

- Useful for generic collection or generic classes representing the items in the collection
- Can use the generic classes with the generic interfaces to avoid boxing and unboxing operator on the value types
- Generic class can implement the generic interfaces by passing the required parameter

Generic Methods

```
using System;
using System.Collections.Generic;
namespace GenericMethodDemo {
   class Program {
      static void Swap<T>(ref T lhs, ref T rhs) {
         T temp;
         temp = lhs;
         lhs = rhs;
         rhs = temp;
      }
      static void Main(string[] args) {
         int a, b;
         char c, d;
         a = 10;
         b = 20;
         c = 'I';
         d = 'V';
         //display values before swap:
         Console.WriteLine("Int values before calling swap: ");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values before calling swap: ");
         Console.WriteLine("c = {0}, d = {1}", c, d);
         // call swap
         Swap<int>(ref a, ref b);
         Swap<char>(ref c, ref d);
         // display values after swap:
         Console.WriteLine("Int values after calling swap: ");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values after calling swap: ");
         Console.WriteLine("c = {0}, d = {1}", c, d);
         Console.ReadKey();
      }
   }
}
```

Iterators

- Iterators used to traverse through a list of values or a collection
- Iterator is not a data member but is a way of accessing the member
- Interator can be a method, a get accessor or an operator that allows to navigate through the values in a collection

Iterators Benefits

- Provide a simplified and faster way of interating through the values of a collection
- Reduce the complexity of providing an enumerator for a collection
- Can return large number of values
- Can be used to evaluate and return only those values that are needed
- Can return values without consuming memory by referring each value in the list

Use Iterators

- Implementation
  - Created by implementing the GetEnumerator() method that returns a reference of the IEnumerator interface:

```
public System.Collections.IEnumerator GetEnumerator()
{
    for (int i = 0; i < 10; i++)
    {
        yield return i;
    }
}
```

- Implementing Named Iterators
  - Another way to creating iterators is by creating a method whose return type is the IEnumerable interface. This is called a Named Iterators:

```
ListClass test = new ListClass();
foreach (int n in test.SampleIterator(1, 10))
{
    System.Console.Write(n + " ");
}
```

Iterators Example 1

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}
```

```
public static System.Collections.Generic.IEnumerable<int>
                EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Iterators Example 2

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();
    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}
```

```
public class DaysOfTheWeek : IEnumerable
{
    private string[] days = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

Summary

- Collection classes are specialized classes for data storage and retrieval
- ArrayList represents an ordered collection of an object that can be indexed individually.
- The Hashtable class represents a collection of **key-and-value pairs** that are organized based on the hash code of the key
- Iterators used to traverse through a list of values or a collection
- **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