.NET Framework 4.7 and C# 8.0

Lesson 09 : Collections and Generics

Lesson Objectives

- In this lesson, you will learn about:
  - Collection classes and collection interfaces in System. Collections Namespace
  - Generics
  - Iterators
  - Collection Initializers

Need for collections

- You are developing a Student-tracking application.
- You implement a data structure named Student to store student information.
- However, you do not know the number of records that you need to maintain.
- You can store the data structure in an array, but then you would need to write code to add each new employee.
- To add a new item to an array, you first have to create a new array that has room for an additional element.

Need for collections

- Then, you need to copy the elements from the original array into the new array and add the new element.
- To simplify this process, the .NET Framework provides classes that are collectively known as Collections.
- By using collections, you can store several items within one object.
- Collections have methods that you can use to add and remove items.
- These methods automatically resize the corresponding data structures without requiring additional code.

What are Collections?

- In C#, collections are:
  - Groups of objects.
  - Enumerable data structures that can be accessed using indexes or keys.
- The .NET Framework:
  - Has powerful support for collections.
  - It contains a large number of interfaces and classes that define and implement various types of collections.

System.Collections namespace

- .NET Framework System.Collections namespace provides:
  - Collection Interfaces:
    - Collection Interfaces define standard methods and properties implemented by different types of data structures.
    - These interfaces allow enumerable types to provide consistency, and aid interoperability.
  - Collection Classes:
    - Functionality-rich implementation of many common collection classes such as lists and dictionaries.
    - These all implement one or more of the common collection interfaces.

ICollection Interface

- ICollection Interface:
  - Is the foundation of the collections namespace and is implemented by all the collection classes.
  - Defines only the most basic collection functionality.
- ICollection Properties:
  - Count: Returns the number of items in the collection.
  - IsSynchronized: Returns true if this instance is thread-safe.
  - SyncRoot: Returns an object that can be used to provide synchronized access to the collection.
- Methods inside ICollection:
  - CopyTo(): Copies all elements in the collection into an array.

IEnumerable Interfaces

- IEnumerable interface:
  - An enumerator is an object that provides a forward, read-only cursor for a set of items.
  - The IEnumerable interface has one method called the GetEnumerator() method.
  - Classes implementing this method must return a class that implements the IEnumerator interface.

IEnumerator Interfaces

- IEnumerator Interface:
  - Defines the notion of a cursor that moves over the elements of a collection.
  - Has three members for moving the cursor and retrieving elements from the collection.
- IEnumerator Properties:
  - Current: It returns the element at the position of the cursor.
- IEnumerator Methods:
  - MoveNext() : This method advances the cursor returning true if the cursor was successfully advanced to the next element and false if the cursor has moved past the last element.

ArrayList Class

- The ArrayList class is a dynamic array of heterogeneous objects.
- In an array we can store only objects of the same type. However, in an ArrayList we can have different types of objects.
- These in turn would be stored as object type only.
- An ArrayList uses its indexes to refer to a particular object stored in its collection.
- ArrayList properties and methods:
  - The Count property gives the total number of items stored in the ArrayList object.
  - The Capacity property gets or sets the number of items that the ArrayList object can contain.
  - Objects are added using the Add() method of the ArrayList and removed using its Remove() method.

ArrayList: Example

- Example:

```
class Test
{
    static void Main()
    {
        int intValue = 100;
        double doubleValue = 20.5;
        ArrayList arrayList = new ArrayList();
            arrayList.Add("John");
            arrayList.Add(intValue);
            arrayList.Add(doubleValue);

            for (int index = 0; index <arrayList.Count; index++)
                Console.WriteLine(arrayList[index]);
    }
}
```

Stack Class

- The Stack Class:
  - Provides a Last-in-First-out (LIFO) collection of items of the System.Object type. The last added item is always at the top of the Stack and is also the first one to be removed.
- Important Operations of the Stack class:
  - Push: Inserts an object at the top of the Stack
  - Pop: Returns and permanently removes the object at the top of the Stack.
  - Peek: Returns the object at the top of the Stack without removing it.
  - Clear: Clears the stack by removing all objects from the Stack.
  - CopyTo: Copies the Stack to an existing one-dimensional Array.

Stack Class: Example

```
class Test
{
    static void Main()
    {
        Stack stackObject = new Stack();
        stackObject.Push("Joydip");
        stackObject.Push("Steve");
        stackObject.Push("Jini");

        while (stackObject.Count > 0)
            Console.WriteLine(stackObject.Pop());
        Console.ReadLine();
    }
}
```

Queue Data Structure

- Queue:
  - Is a data structure that provides a First-in-First-out collection of items of the System.Object type.
- In the Queue:
  - Newly added items are stored at the end or the rear of the Queue and items are deleted from the front of the Queue.
- The Enqueue() method stores items at rear of the Queue
- The Dequeue() method removes items from front of the Queue.

Queue: Example

```
class Test
{
    static void Main()
    {
        Queue queueObject = new Queue();
        queueObject.Enqueue("Joydip");
        queueObject.Enqueue("Steve");
        queueObject.Enqueue("Jini");

        while (queueObject.Count > 0)
            Console.WriteLine(queueObject.Dequeue());
        Console.ReadLine();
    }
}
```

Hashtable Class

- The Hashtable Class:
  - Creates a collection that uses a hash table for storage.
  - Represents a dictionary of associated keys and values, implemented as a hash table.
  - Provides a faster way of storage and retrieval of items of the object type.
  - Provides support for key based searching.
- The GetHashCode() method of the Hashtable class returns the hash code for an object instance.

Hashtable Class: Example

```
class Test
{
        static void Main()
    {
        Hashtable hashTable = new Hashtable();
        hashTable.Add(1, "Joydip");
        hashTable.Add(2, "Manashi");
        hashTable.Add(3, "Jini");
        hashTable.Add(4, "Piku");
        Console.WriteLine("The keys and values are:");
        foreach (int k in hashTable.Keys)
        {
                Console.WriteLine(k);
                Console.WriteLine(hashTable[k].ToString());
        }
    }
}
```

Demo

- Demo on Collection Classes

Why Generics?

- Without generics, general-purpose data structures can use type object to store data of any type.

```
public class Stack
{
    object[] items;
    int count;
    public void Push(object item) {...}
    public object Pop() {...}
}
```

Why Generics? (Cont..)

- To push a value of any type, such as a Customer instance, onto a stack.

```
Stack stack = new Stack();
stack.Push(new Customer())
```

- However, when a value is retrieved, the result of the Pop method must explicitly be cast back to the appropriate type,

```
Customer c = (Customer)stack.Pop();
```

- This is tedious to write and carries a performance penalty for runtime type checking.

Why Generics? (Cont..)

- Similarly, if a value of a value type, such as an int, is passed to the Push method, it is automatically boxed.
- When the int is later retrieved, it must be unboxed with an explicit type cast.

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

- Such boxing and unboxing operations add performance overhead because they involve dynamic memory allocations and runtime type checks.

What is Generics?

- Generics provide a facility for creating types that have type parameters.
- Following example declares a generic Stack class with a type parameter T:

```
public class Stack<T>
{
    T[ ] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
}
```

What is Generics? (cont..)

- The type parameter is specified in < and > delimiters after the class name.
- The type parameter T acts as a placeholder until an actual type is specified at use.
- In the following example, int is given as the type argument for T:

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

What is Generics? (cont..)

- Similarly we can have:

```
Stack<Customer> objStack = new
Stack<Customer>();
objStack.Push(new Customer());
Customer objCust = objStack.Pop();
```

What is Generics? (cont..)

- Generic type declarations may have any number of type parameters. The Stack<T> example in the previous slide has only one type parameter.

- For example, a generic Dictionary class might have two type parameters, one for the type of the keys and one for the type of the values

```
public class Dictionary<K,V>
{
public void Add(K key, V value) {...}
public V this[K key] {...}
}
```

What is Generics? (cont..)

- When Dictionary<K,V> is used, two type arguments would have to be supplied:

```
Dictionary<string,Customer> objDict = new Dictionary<string,Customer>();
objDict.Add("Peter", new Customer());
Customer objCust = objDict["Peter"];
```

Demo

- Demo on Generics

Constraints

- A generic class will do more than just store data based on a type parameter - the generic class will want to invoke methods on objects whose type is given by a type parameter.
- Example: An Add method in a Dictionary<K,V> class might need to compare keys using a CompareTo method.

```
 public class Dictionary<K,V>
{    public void Add(K key, V value)
     {
     ...
     if (key.CompareTo(x) < 0) {...} // Error, no CompareTo
     method
     ...
     }
}
```

Constraints (Cont..)

- To provide stronger compile-time type checking and reduce type casts, C# permits an optional list of. constraints to be supplied for each type parameter.
- A type parameter constraint specifies a requirement that a type must fulfill in order to be used as an argument for that type parameter.
- Constraints are declared using the word where, followed by the name of a type parameter, followed by a list of class or interface types and optionally the constructor constraint new().

Constraints (Cont..)

- For the Dictionary<K,V> class to ensure that keys always implement IComparable, the class declaration can specify a constraint for the type parameter K.

```
public class Dictionary<K,V> where K: IComparable
{
    public void Add(K key, V value)
    {
            ...
            if (key.CompareTo(x) < 0) {...}
            ...
    }
}
```

Constraints (Cont..)

- Given the above declaration, the compiler will ensure that any type argument supplied for K is a type that implements IComparable
- For a given type parameter, it is possible to specify any number of interfaces as constraints, but no more than one class

```
public class EntityTable<K,E>
where K: IComparable<K>, IPersistable
where E: Entity, new()
{
    public void Add(K key, E entity)
    {         ...
              if (key.CompareTo(x) < 0) {...}
              ...
    }
}
```

Demo

- Demo on constraints

Generic Methods

- A type parameter may not be needed for an entire class but is needed only inside a particular method

```
void PushMultiple(Stack<int> stack, params int[] values)
{
    foreach (int value in values)
            stack.Push(value);
}
```

- The above method can be used to push multiple int values

```
Stack<int> stack = new Stack<int>();
PushMultiple( stack, 1, 2, 3, 4);
```

Generic Methods (cont..)

- The previous method works with the particular constructed type Stack<int> only.
- To have it work with any Stack<T>, the method must be written as a generic method.
- A generic method has one or more type parameters specified in < and > delimiters after the method name.
- A generic PushMultiple method:

```
void PushMultiple<T>(Stack<T> stack, params T[] values)
{
    foreach (T value in values)
            stack.Push(value);
}
```

Generic Methods (cont..)

- When calling a generic method, type arguments are given in angle brackets in the method invocation

- Example:

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

Demo

- Demo on Generic Method

Generic Interfaces

- Often used to define interfaces either for generic Collection classes, or for the generic classes that represent items in the Collection
- Preferable to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types.
- Generic Interfaces inside System.Collections.Generic:
  - ICollection<T>
  - IComparable<T>
  - IEnumerable<T>
  - IEnumerator<T>
  - IComparer<T>

What are Iterators?

- An iterator is a method, get accessor or operator that enables you to support foreach iteration in a class or struct without having to implement the entire IEnumerable interface.
- An iterator is a section of code that returns an ordered sequence of values of the same type.
- The iterator code uses the yield return statement to return each element in turn, yield break ends the iteration.
- When the compiler detects an iterator, it automatically generates the Current(), MoveNext() and Dispose() methods of the IEnumerable or IEnumerable<T> interface.
- The return type of an iterator must be IEnumerable, IEnumerator, IEnumerable<T>, or IEnumerator<T>
- Iterators are especially useful with collection classes.

The yield Statement

- yield Statement:
  - Is used in an iterator block to provide a value to the enumerator object or to signal the end of iteration.
  - It takes one of the following forms:
    - yield return <expression>;
    - yield break;
  - expression is evaluated and returned as a value to the enumerator object
  - expression has to be implicitly convertible to the yield type of the iterator

The yield Statement: Example

```
public class DaysOfTheWeek :
System.Collections.IEnumerable
{
     string[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr",
"Fri", "Sat"};
     public System.Collections.IEnumerator GetEnumerator()
     {
          for (int i = 0; i < m_Days.Length; i++)
          {
               yield return m_Days[i];
          }
     }
}
```

Iterators: Example

```
class TestDaysOfTheWeek
{
    static void Main()
    {
        DaysOfTheWeek week = new DaysOfTheWeek();
        foreach (string day in week)
        {
            System.Console.Write(day + " ");
        }
    }
}
```

Demo

- Demo on Iterators

What are Collection Initializers?

- Any object that implements IEnumerable<T> and has a public Add method can have its values initialized with a collection initializer
- A collection initializer consists of a sequence of element initializers, enclosed by { and } tokens and separated by commas.
- Example:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 6, 8, 9 };
```

Collection Initializers (Cont..)

- Creating a shape that is made up of a collection of points:

```
List<Point> Square = new List<Point>
{
    new Point { X=0, Y=5 },
    new Point { X=5, Y=5 },
    new Point { X=5, Y=0 },
    new Point { X=0, Y=0 }
};
```

Summary

- In this module, we explored System.Collections namespace and the collection interfaces and classes present in it.
- These collection Interfaces and classes are:

Collection Interfaces	**Collection Classes**
ICollection	ArrayList
IEnumerable	Stack
IEnumerator	Queue
	BitArray
	HashTable

Review Question

- What are the advantages of an ArrayList? How is it different from an Array?
- What is the use of the IEnumerable interface?
- What are the different operations possible with a BitArray Class?
- What is the difference between pop and peek method of a Stack class?
- What is the need for Generics?
- Can Delegates also be made Generic?