C# 7.0

Lesson 09 : Collections & Generics

Lesson Objectives

- Need for Collections
- What are Collections?
- System.Collections Namespace
- ICollection Interface
- IEnumerable Interfaces
- IEnumerator Interfaces
- ArrayList Collection
- ArrayList Collection – An Example
- Why Generics?
- Concept of Generics in C#

Need for Collections

- You are developing an 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 (Cont.)

- 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 Collection

- 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 Collection – An 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]);
     }
}
```

Demo

- Demo on Implementing Collections in C#

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.

Concept of Generics in C#

- 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() {...}
}
```

Concept of Generics in C#(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();
```

Concept of Generics in C# (Cont.)

- Similarly we can have:

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

Concept of Generics in C# (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] {...}
}
```

Concept of Generics in C# (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 Implementing Generics in C#

Summary

In this lesson, you have learnt

- 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
  - ICollection
  - IEnumerable
  - Ienumerator
- Collection Classes
  - ArrayList
  - Stack
  - Queue
  - BitArray
  - HashTable

Review Question

- Question 1: What are the advantages of an ArrayList? How is it different from an Array?
- Question 2: What is the use of the IEnumerable interface?
- Question 3: What is the difference between pop and peek method of a Stack class?
- Question 4: What is the need for Generics?