Dictionary Collection in C#

A C# dictionary is a generic collection of key-value pairs. The keys must be unique, but the values can be duplicated. Dictionaries are implemented as hash tables so their keys can quickly access them.

To create a dictionary in C#, you use the Dictionary<TKey, TValue> class, where TKey is the type of the keys, and TValue is the type of the values. For example, the following code creates a dictionary that can store strings as keys and integers as values:

- Dictionary<string, int> dictionary = new Dictionary<string, int>();
- Dictionary class is present in System.Collections.Generic namespace.
- When creating a dictionary, we need to specify the type for key and value.
- Dictionary provides fast lookups for values using keys.
- Keys in the dictionary must be unique.

Basic operation in dictionary

- Add Elements
- Access Elements
- Change Elements
- Remove Elements

Add() method using which we can add elements in the dictionary.

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // create a dictionary
        Dictionary<string, string> mySongs = new Dictionary<string, string>();

        // add items to dictionary
        mySongs.Add("Queen", "Break Free");
        mySongs.Add("Free", "All right now");
        mySongs.Add("Pink Floyd", "The Wall");

    }
}
```

accessed the values of the dictionary using their keys:

student["Name"] - accesses the value whose key is "Name"

student["Faculty"] - accesses the value whose key is "Faculty"

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // create a dictionary
        Dictionary<string, string> student = new Dictionary<string, string>();

        // add items to dictionary
        student.Add("Name", "Susan");
        student.Add("Faculty", "History");

        // access the value having key "Name"
        Console.WriteLine(student["Name"]);

        // access the value having key "Faculty"
        Console.WriteLine(student["Faculty"]);
    }
}
```

Change element in dictionary

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // create a dictionary
        Dictionary<string, string> car = new Dictionary<string, string>();

        // add items to dictionary
        car.Add("Model", "Hyundai");
        car.Add("Price", "36K");

        // print the original value
        Console.WriteLine("Value of Model before changing: " + car["Model"]);

        // change the value of "Model" key to "Maruti"
        car["Model"] = "Maruti";

        // print new updated value of "Model"
        Console.WriteLine("Value of Model after changing: " + car["Model"]);
    }
}
```

To remove the elements inside the dictionary

Remove() - removes the key/value pair from the dictionary

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // create a dictionary
        Dictionary<string, string> employee = new Dictionary<string, string>();

        // add items to dictionary
        employee.Add("Name", "Marry");
        employee.Add("Role", "Manager");
        employee.Add("Address", "California");

        Console.WriteLine("Original Dictionary :");

        // iterate through the modified dictionary
        foreach (KeyValuePair<string, string> items in employee)
        {
            Console.WriteLine("{0} : {1}", items.Key, items.Value);
        }

        // remove value with key "Role"
        employee.Remove("Role");

        Console.WriteLine("\nModified Dictionary :");

        // iterate through the modified dictionary
        foreach (KeyValuePair<string, string> items in employee)
        {
            Console.WriteLine("{0} : {1}", items.Key, items.Value);
        }
    }
}
```

Advanced features in dictionary

- TryGetValue() - get value from dictionary
- Count() - Gets element count
- Remove() - Remove element passing key
- Clear() - remove all elements from dict
- Using LINQ extension methods with Dictionary

Example of Dictionary

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

namespace ConsoleApplication.Day_10.DictionaryExample
{
    public class DictionaryDemo
    {
        public static void DictionaryDemoMethod()
        {
            Customer customer1 = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500, CustomerType = "Veg" };
            Customer customer2 = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350, CustomerType = "Veg" };
            Customer customer3 = new Customer() { Id = 103, Name = "Customer Three",
BillAmount = 800, CustomerType = "Veg" };
            Customer customer4 = new Customer() { Id = 104, Name = "Customer Four",
BillAmount = 900, CustomerType = "NonVeg" };
            Customer customer5 = new Customer() { Id = 105, Name = "Customer Five",
BillAmount = 1200, CustomerType = "NonVeg" };

            Dictionary<int, Customer> dictionaryOfCustomers = new Dictionary<int,
Customer>();
            dictionaryOfCustomers.Add(customer1.Id, customer1);
            dictionaryOfCustomers.Add(customer2.Id, customer2);
            dictionaryOfCustomers.Add(customer3.Id, customer3);
            dictionaryOfCustomers.Add(customer4.Id, customer4);
            dictionaryOfCustomers.Add(customer5.Id, customer5);
            // dictionaryOfCustomers.Add(customer5.Id, customer5);

            Customer[] custArray = new Customer[5];
            custArray[0] = customer1;
            custArray[1] = customer2;
            custArray[2] = customer3;
            custArray[3] = customer4;
            custArray[4] = customer5;

            //array to dictionary
            Console.WriteLine("Dictionary Elements");
            Dictionary<int, Customer> customersDictionary =
custArray.ToDictionary(customer => customer.Id, customer => customer);
            foreach (KeyValuePair<int, Customer> kvp in customersDictionary)
            {
                Console.WriteLine($"Key = {kvp.Key}");
                Customer customer33 = kvp.Value;
                Console.WriteLine($"Id = {customer33.Id} Name = {customer33.Name}
Bill Amount = {customer33.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            Console.WriteLine("BEFORE Clear");
            foreach (KeyValuePair<int, Customer> customer in dictionaryOfCustomers)
            {
                Console.WriteLine("Key = {0}", customer.Key);
                Customer cust = customer.Value;
```

```
                Console.WriteLine($"Id = {cust.Id} ; Name = {cust.Name} ; Type =
{cust.CustomerType} ; Bill Amount = {cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }
            dictionaryOfCustomers.Remove(200);
            dictionaryOfCustomers.Clear();
            Console.WriteLine("AFTER Clear");
            foreach (KeyValuePair<int, Customer> customer in dictionaryOfCustomers)
            {
                Console.WriteLine("Key = {0}", customer.Key);
                Customer cust = customer.Value;
                Console.WriteLine($"Id = {cust.Id} ; Name = {cust.Name} ; Type =
{cust.CustomerType} ; Bill Amount = {cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }
            Console.WriteLine("Count of elements in dictionary : {0}",
dictionaryOfCustomers.Count);

            Customer custValue;
            if (dictionaryOfCustomers.TryGetValue(200, out custValue))
            {
                Console.WriteLine($"Id = {custValue.Id} ; Name = {custValue.Name} ;
Type = {custValue.CustomerType} ; Bill Amount = {custValue.BillAmount}");
            }
            else
            {
                Console.WriteLine("No record found");
            }

            foreach (KeyValuePair<int, Customer> customer in dictionaryOfCustomers)
            {
                Console.WriteLine("Key = {0}", customer.Key);
                Customer cust = customer.Value;
                Console.WriteLine($"Id = {cust.Id} ; Name = {cust.Name} ; Type =
{cust.CustomerType} ; Bill Amount = {cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            //var cust1 = dictionaryOfCustomers[0];
            if (dictionaryOfCustomers.ContainsKey(200))
            {
                var cust1 = dictionaryOfCustomers[101];
                Console.WriteLine($"Id = {cust1.Id} ; Name = {cust1.Name} ; Type =
{cust1.CustomerType} ; Bill Amount = {cust1.BillAmount}");
            }
            else
            {
                Console.WriteLine("No record found");
            }

            Console.ReadLine();
        }
    }
```

```
    class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int BillAmount { get; set; }
        public string CustomerType { get; set; }
    }
}
```

List Collection Class in C#

List<T> class represents the list of objects which can be accessed by index. It comes under the **System.Collections.Generic** namespace. List class can be used to create a collection of different types like integers, strings etc. List<T> class also provides the methods to search, sort, and manipulate lists.

There are several generic collection classes in System.Collections.Generic namespace as listed below:

1. Dictionary
2. List
3. Stack
4. Queue

Characteristics:

- It is different from the arrays. A **List<T> can be resized dynamically** but arrays cannot.
- List<T> class can accept null as a valid value for reference types and it also allows duplicate elements.
- If the Count becomes equals to Capacity, then the capacity of the List increased automatically by reallocating the internal array. The existing elements will be copied to the new array before the addition of the new element.
- List<T> class is the generic equivalent of ArrayList class by implementing the IList<T> generic interface.
- This class can use both equality and ordering comparer.
- List<T> class is not sorted by default and elements are accessed by zero-based index.
- A List class can be used to create a collection of any type.
- For example, we can create a list of Integers, Strings and even complex types.
- Unlike arrays, lists can grow in size automatically.
- This class also provides methods to search, sort, and manipulate lists.

- **Contains() function** - Use this function to check if an item exists in the list. This method returns true if the items exists, else false.
- **Exists() function** - Use this function, to check if an item exists in the list based on a condition. This method returns true if the items exists, else false.
- **Find() function** - This method searches for an element that matches the conditions defined by the specified lambda expression and returns the first matching item from the list.
- **FindLast() function -** This method searches for an element that matches the conditions defined by the specified lambda expression and returns the Last matching item from the list.
- **FindAll() function** - This method returns all the items from the list that match the conditions specified by the lambda expression.

- **FindIndex() function** - This method returns the index of the first item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.

- **FindLastIndex() function** - This method returns the index of the last item, that matches the condition specified by the lambda expression. There are 2 other overloads of this method which allows us to specify the range of elements to search, with in the list.

- **Convert an array to a List** - Use ToList() method 9. Convert a list to an array - Use ToArray() method 10. Convert a List to a Dictionary - Use ToDictionary() method.

- AddRange() - Add() method allows you to add one item at a time to the end of the list, where as AddRange() allows you to add another list of items, to the end of the list. 2.

- GetRange() - Using an item index, we can retrieve only one item at a time from the list, if you want to get a list of items from the list, then use GetRange() function. This function expects 2 parameters, i.e the start index in the list and the number of elements to return. 3.

- InsertRange() - Insert() method allows you to insert a single item into the list at a specified index, where as InsertRange() allows you, to insert another list of items to your list at the specified index. 4.

- RemoveRange() - Remove() function removes only the first matching item from the list.

- RemoveAt() function, removes the item at the specified index in the list.

- RemoveAll() function removes all the items that matches the specified condition.

RemoveRange() method removes a range of elements from the list. This function expects 2 parameters, i.e the start index in the list and the number of elements to remove. If you want to remove all the elements from the list without specifying any condition, then use Clear() function

Sort a list a simple type

Sorting in c# is the process of arranging the contents of a collection in a specific order. A collection may be an array, a list or any other data group. The collection may contain elements of simple types as well as complex types. A simple type may be a collection of integers, strings, floating-point numbers, etc. A complex type may be a collection of objects of user-defined types such as Employee, Student, etc. Complex types are more than often nested, meaning the objects may have multiple attributes.

- **List<T>.Sort() Method** is used to sort the elements or a portion of the elements in the List<T> using either the specified or default IComparer<T> implementation or a provided Comparison<T> delegate to compare list elements.
- List numbers = new List { 1, 8, 7, 5, 2, 3, 4, 9, 6 }; numbers.Sort();

- If you want the data to be retrieved in descending order, use Reverse() method on the list instance. numbers.Reverse();

However, when you do the same thing on a complex type like Customer, we get a runtime invalid operation exception - Failed to compare 2 elements in the array. This because, .NET runtime does not know, how to sort complex types. We have to tell the way we want data to be sorted in the list by implementing IComparable interface

Exceptions:

- **InvalidOperationException:** If the comparer is null, and the default comparer Default cannot find the implementation of the IComparable<T> generic interface or the IComparable interface for type T.

- **ArgumentException:** If the implementation of comparer caused an error during the sort. For example, comparer might not return 0 when comparing an item with itself.

Simple sorting type Example

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

namespace ConsoleApplication.Day_10.ListSortOperationExamples
{
    public class ListSortOperations
    {
        public static void ListOperations()
        {
            List<int> intList = new List<int>();
            intList.Add(1);
            intList.Add(6);
            intList.Add(4);
            intList.Add(5);
            intList.Add(3);
            intList.Add(2);

            List<string> stringList = new List<string>();
            stringList.Add("F");
            stringList.Add("C");
            stringList.Add("A");
            stringList.Add("D");
            stringList.Add("B");
            stringList.Add("E");

            Console.WriteLine("Before Sort Operation");
            foreach (string i in stringList)
            {
                Console.Write(i + " ");
            }

            stringList.Sort();

            Console.WriteLine("\n");
            Console.WriteLine("After Sort Operation in asending");
            foreach (string i in stringList)
            {
                Console.Write(i + " ");
            }

            stringList.Reverse();
            Console.WriteLine("\n");
            Console.WriteLine("After Sort Operation in descending");
            foreach (string i in stringList)
            {
                Console.Write(i + " ");
            }

            Customer customer1 = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500, CustomerType = "Veg" };
            Customer customer2 = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350, CustomerType = "Veg" };
```

```
            Customer customer3 = new Customer() { Id = 103, Name = "Customer Three",
BillAmount = 800, CustomerType = "Veg" };
            Customer customer4 = new Customer() { Id = 104, Name = "Customer Four",
BillAmount = 900, CustomerType = "NonVeg" };
            Customer customer5 = new Customer() { Id = 105, Name = "Customer Five",
BillAmount = 1200, CustomerType = "NonVeg" };

            List<Customer> customers = new List<Customer>();
            customers.Add(customer5);
            customers.Add(customer4);
            customers.Add(customer2);
            customers.Add(customer3);
            customers.Add(customer1);

            Console.WriteLine("Before Sort");
            foreach (Customer cust in customers)
            {
                Console.WriteLine($"Id = {cust.Id} ; Name = {cust.Name} ; Bill
Amount = {cust.BillAmount} ; Customer Type = {cust.CustomerType}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            customers.Sort();

            Console.WriteLine("After Sort");
            foreach (Customer cust in customers)
            {
                Console.WriteLine($"Id = {cust.Id} ; Name = {cust.Name} ; Bill
Amount = {cust.BillAmount} ; Customer Type = {cust.CustomerType}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            Console.ReadLine();
        }
    }

    class Customer : IComparable<Customer>
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int BillAmount { get; set; }
        public string CustomerType { get; set; }

        public int CompareTo(Customer obj)
        {
            if (this.BillAmount > obj.BillAmount)
                return 1;
            else if (this.BillAmount < obj.BillAmount)
                return -1;
            else
                return 0;
        }
    }
}
```

Sort a list of complex types in C#

To sort a list of complex types without using LINQ, the complex type has to implement the **IComparable** interface and needs to provide the implementation for the **CompareTo()** method as follows. The **CompareTo()** method returns an integer value and the meaning of the return value as shown below.

1. **Return value greater than ZERO** – The current instance is greater than the object being compared with.
2. **Return value less than ZERO** – The current instance is less than the object being compared with.
3. **The Return value is ZERO** – The current instance is equal to the object being compared with.

Alternatively, we can also invoke the CompareTo() method directly. The Salary property of the Employee object is int and the CompareTo() method is already implemented on the integer type that we already discussed, so we can invoke this method and return its value.

- **Sort():** This method is used to sort the elements in the entire Generic List using the default comparer.

- **Sort(IComparer<T> comparer):** This method is used to sort the elements in the entire Generic List using the specified comparer.

- **Sort(Comparison<T> comparison):** This method is used to sort the elements in the entire Generic List using the specified System.Comparison.

- **Sort(int index, int count, IComparer<T> comparer):** This method is used to sort the elements in a range of elements in a Generic List using the specified comparer.

Example of complex sorting types

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

namespace ConsoleApplication.Day_10.ListOfComplexTypeExample
{
    public class ListOfComplexType
    {
        public static void ListExample()
        {
            Customer customer1 = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500 };
            Customer customer2 = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350 };
            Customer customer3 = new Customer() { Id = 103, Name = "Customer Three",
BillAmount = 800 };
            Customer customer4 = new Customer() { Id = 104, Name = "Customer Four",
BillAmount = 900 };

            Customer[] customersArray = new Customer[2];
            customersArray[0] = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500 };
            customersArray[1] = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350 };
            //customersArray[2] = new Customer() { Id = 103, Name = "Customer
Three", BillAmount = 800 };

            Console.WriteLine("Array Output");
            foreach (Customer cust in customersArray)
            {
                Console.WriteLine($"Id = {cust.Id} Name = {cust.Name} Bill Amount =
{cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            Console.WriteLine("List Output");
            List<Customer> customers = new List<Customer>(2);
            customers.Add(customer1);
            customers.Add(customer2);
            customers.Add(customer3);
            customers.Add(customer4);

            //insert elect at specific index - Insert()
            customers.Insert(2, customer4);
            foreach (Customer cust in customers)
            {
                Console.WriteLine($"Id = {cust.Id} Name = {cust.Name} Bill Amount =
{cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            Console.WriteLine($"Index of Customer 4 =
{customers.IndexOf(customer4)}");

            //Accessing list element by index
            Customer cust1 = customers[2];
```

```
            Console.WriteLine($"Id = {cust1.Id} Name = {cust1.Name} Bill Amount =
{cust1.BillAmount}");

            //Printing all elements from array
            foreach (Customer cust in customers)
            {
                Console.WriteLine($"Id = {cust.Id} Name = {cust.Name} Bill Amount =
{cust.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------");
            }

            Console.ReadLine();
        }
    }

    class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int BillAmount { get; set; }
    }
}
```

Generic queue collection class

- **A Queue is a First in First Out (FIFO) collection class in the System.Collections.**Generic class that is present in System.Collections.Generic namespace. The Queue collection class is analogous to a queue at the ATM machine to withdraw money. The order in which people queue up, will be the order in which they will be able to get out of the queue and withdraw money from the ATM. The Queue collection class operates in a similar fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed (dequeued) from the Queue.

- To add items to the end of the queue, use **Enqueue()** method.

- To remove an item that is present at the beginning of the queue, use **Dequeue()** method.

- A **foreach loop** iterates thru the items in the queue, but will not remove them from the queue

- To check if an item, exists in the queue, use **Contains()** method.

- difference between **Dequeue()** and **Peek()** methods.

- **Dequeue()** method removes and returns the item at the beginning of the queue, where as **Peek()** returns the item at the beginning of the queue, without removing it.

Example of Generic queue collection class

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

namespace ConsoleApplication.Day_10.QueueCollectionExample
{
    public class QueueCollection
    {
        public static void QueueMainMethod()
        {
            Customer customer1 = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500, CustomerType = "Veg" };
            Customer customer2 = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350, CustomerType = "Veg" };
            Customer customer3 = new Customer() { Id = 103, Name = "Customer Three",
BillAmount = 800, CustomerType = "Veg" };
            Customer customer4 = new Customer() { Id = 104, Name = "Customer Four",
BillAmount = 900, CustomerType = "NonVeg" };
            Customer customer5 = new Customer() { Id = 105, Name = "Customer Five",
BillAmount = 1200, CustomerType = "NonVeg" };

            Queue<Customer> queueOfCustomers = new Queue<Customer>();
            queueOfCustomers.Enqueue(customer1);
            queueOfCustomers.Enqueue(customer2);
            queueOfCustomers.Enqueue(customer3);
            queueOfCustomers.Enqueue(customer4);
            queueOfCustomers.Enqueue(customer5);

            foreach (Customer customer in queueOfCustomers)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            Console.WriteLine("\n");
            Console.WriteLine("********************** Dequeue
***********************************");
            Customer cust1 = queueOfCustomers.Dequeue();
            Console.WriteLine($"Id = {cust1.Id} ; Name = {cust1.Name} ; Type =
{cust1.CustomerType} ; Bill Amount = {cust1.BillAmount}");

            Console.WriteLine("\nAfter Dequeue");

Console.WriteLine("*****************************************************************
****************");
            foreach (Customer customer in queueOfCustomers)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
```

```
            }

            Console.WriteLine("\n");
            Console.WriteLine("********************** Peek
**********************************");
            Customer cust2 = queueOfCustomers.Peek();
            Console.WriteLine($"Id = {cust2.Id} ; Name = {cust2.Name} ; Type =
{cust2.CustomerType} ; Bill Amount = {cust2.BillAmount}");
            Customer cust3 = queueOfCustomers.Peek();
            Console.WriteLine($"Id = {cust3.Id} ; Name = {cust3.Name} ; Type =
{cust3.CustomerType} ; Bill Amount = {cust3.BillAmount}");
            Customer cust4 = queueOfCustomers.Peek();
            Console.WriteLine($"Id = {cust4.Id} ; Name = {cust4.Name} ; Type =
{cust4.CustomerType} ; Bill Amount = {cust4.BillAmount}");

            Console.WriteLine("\nAfter Peek");

Console.WriteLine("**************************************************************
****************");
            foreach (Customer customer in queueOfCustomers)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            Console.WriteLine("\n");
            Console.WriteLine("Count of elements in Queue {0}",
queueOfCustomers.Count);

            if (queueOfCustomers.Contains(customer1))
            {
                Console.WriteLine("Customer 5 is available.");
            }
            else
            {
                Console.WriteLine("Not available");
            }

            Console.ReadLine();
        }
    }
    class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int BillAmount { get; set; }
        public string CustomerType { get; set; }
    }
}
```

Generic stack collection class

- Stack is a generic LIFO (Last In First Out) collection class that is present in System.Collections.Generic namespace. The Stack collection class is analogous to a stack of plates. If you want to add a new plate to the stack of plates, you place it on top of all the already existing plates. If you want to remove a plate from the stack, you will first remove the one that you have last added. The stack collection class also operates in a similar fashion. The last item to be added (pushed) to the stack, will be the first item to be removed (popped) from the stack.
- To insert an item at the top of the stack, use **Push()** method.
- To remove and return the item that is present at the top of the stack, use **Pop()** method.
- A foreach loop iterates thru the items in the stack, but will not remove them from the stack. The items from the stack are retrieved in **LIFO (Last In First Out),** order. The last element added to the Stack is the first one to be removed.
- To check if an item exists in the stack, use **Contains()** method.
- difference between **Pop() and Peek()** methods? Pop() method removes and returns the item at the top of the stack, where as **Peek()** returns the item at the top of the stack, without removing it.

Example of Generic stack collection class

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

namespace ConsoleApplication.Day_10.StackCollectionExample
{
    public class StackCollection
    {
        public static void StackDemo()
        {
            Customer customer1 = new Customer() { Id = 101, Name = "Customer One",
BillAmount = 500, CustomerType = "Veg" };
            Customer customer2 = new Customer() { Id = 102, Name = "Customer Two",
BillAmount = 350, CustomerType = "Veg" };
            Customer customer3 = new Customer() { Id = 103, Name = "Customer Three",
BillAmount = 800, CustomerType = "Veg" };
            Customer customer4 = new Customer() { Id = 104, Name = "Customer Four",
BillAmount = 900, CustomerType = "NonVeg" };
            Customer customer5 = new Customer() { Id = 105, Name = "Customer Five",
BillAmount = 1200, CustomerType = "NonVeg" };

            Stack<Customer> stackOfCustomer = new Stack<Customer>();
            stackOfCustomer.Push(customer1);
            stackOfCustomer.Push(customer2);
            stackOfCustomer.Push(customer3);
            stackOfCustomer.Push(customer4);
            stackOfCustomer.Push(customer5);

            foreach (Customer customer in stackOfCustomer)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            if (stackOfCustomer.Contains(customer5))
            {
                Console.WriteLine("Customer 5 is available.");
            }
            else
            {
                Console.WriteLine("Not available");
            }

            Console.WriteLine("\n");
            Console.WriteLine("********************** POP
***********************************");
            Customer cust1 = stackOfCustomer.Pop();
            Console.WriteLine($"Id = {cust1.Id} ; Name = {cust1.Name} ; Type =
{cust1.CustomerType} ; Bill Amount = {cust1.BillAmount}");

            Console.WriteLine("\n");
```

```
Console.WriteLine("**********************************************************************
****************");
            Console.WriteLine("\n");
            Console.WriteLine("Stack after POP");
            foreach (Customer customer in stackOfCustomer)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            Console.WriteLine("\n");
            Console.WriteLine("********************** PEEK
************************************");
            Customer cust2 = stackOfCustomer.Peek();
            Console.WriteLine($"Id = {cust2.Id} ; Name = {cust2.Name} ; Type =
{cust2.CustomerType} ; Bill Amount = {cust2.BillAmount}");
            Customer cust3 = stackOfCustomer.Peek();
            Console.WriteLine($"Id = {cust3.Id} ; Name = {cust3.Name} ; Type =
{cust3.CustomerType} ; Bill Amount = {cust3.BillAmount}");
            Customer cust4 = stackOfCustomer.Peek();
            Console.WriteLine($"Id = {cust4.Id} ; Name = {cust4.Name} ; Type =
{cust4.CustomerType} ; Bill Amount = {cust4.BillAmount}");

            Console.WriteLine("\n");

Console.WriteLine("**********************************************************************
****************");
            Console.WriteLine("\n");
            Console.WriteLine("Stack after PEEK");
            foreach (Customer customer in stackOfCustomer)
            {
                Console.WriteLine($"Id = {customer.Id} ; Name = {customer.Name} ;
Type = {customer.CustomerType} ; Bill Amount = {customer.BillAmount}");
                Console.WriteLine("-----------------------------------------------
---------------------------------");
            }

            Console.ReadLine();
        }
    }
    class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int BillAmount { get; set; }
        public string CustomerType { get; set; }
    }
}
```