DICTIONARY

Dictionary is a generic collection which is usually used to store key/value pairs.

The working of dictionary is quite similar to the non-generic hashtable. The advantage of dictionary is, it is generic type.

Dictionary is defined under **System.Collection.Generic** namespace. It is dynamic in nature meaning the size of the dictionary grows according to the need.

Important Points

- The dictionary class implements the
    - Dictionary<TKey,TValue> Interface
    - IReadOnlyCollection<KeyValuePair<TKey,TValue>> Interface
    - IReadOnlyDictionary<TKey,TValue> Interface
    - IDictionary Interface
- In Dictionary, the key cannot be null, but value can be.
- In Dictionary, the key must be unique. Duplicate keys are not allowed if you try to use duplicate key then compiler will throw an exception.
- In Dictionary, you can only store the same types of elements.
- The capacity of a Dictionary is the number of elements that Dictionary can hold.

Creating a Dictionary

Dictionary class has *7 constructors* which are used to create the Dictionary, here we only use Dictionary<TKey, TValue>() constructor and if you want to learn more about constructors then refer C# | Dictionary Class.

Dictionary<TKey, TValue>(): This constructor is used to create an instance of the Dictionary<TKey, TValue> class that is empty, has the default initial capacity, and uses the default equality comparer for the key type as follows:

Step 1: IncludeSystem.Collection.Generic namespace in your program with the help of using keyword.

Syntax:

```
using System.Collection.Generic;
```

Step 2: Create a Dictionary using Dictionary<TKey, TValue> class as shown below:

```
Dictionary dictionary_name = new Dictionary();
```

Step 3: If you want to add elements in your Dictionary then use Add() method to add key/value pairs in your Dictionary. And you can also add key/value pair in the dictionary without using Add method. As shown in the below example.

Step 4: The key/value pair of the Dictionary is accessed using **three** different ways:

- **for loop:** You can use for loop to access the key/value pairs of the Dictionary.

Example:

```
for(int x=0; i< My_dict1.Count; x++)
{
    Console.WriteLine("{0} and {1}", My_dict1.Keys.ElementAt(x),
                        My_dict1[ My_dict1.Keys.ElementAt(x)]);
}
```

- **Using Index:** You can access individual key/value pair of the Dictionary by using its index value. Here, you just specify the key in the index to get the value from the given dictionary, no need to specify the index. Indexer always takes the key as a parameter, if the given key is not available in the dictionary, then it gives KeyNotFoundException.

Example:

```
Console.WriteLine("Value is:{0}", My_dicti[1123]);
Console.WriteLine("Value is:{0}", My_dicti[1125]);
```

- **foreach loop:** You can use foreach loop to access the key/value pairs of the dictionary.As shown in the below example we access the Dictionary using a foreach loop.

Example:

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

class GFG {

        // Main Method
    static public void Main () {

        // Creating a dictionary
        // using Dictionary<TKey,TValue> class
        Dictionary<int, string> My_dict1 =
                        new Dictionary<int, string>();

          // Adding key/value pairs
          // in the Dictionary
          // Using Add() method
          My_dict1.Add(1123, "Welcome");
          My_dict1.Add(1124, "to");
          My_dict1.Add(1125, "GeeksforGeeks");

          foreach(KeyValuePair<int, string> ele1 in My_dict1)
          {
              Console.WriteLine("{0} and {1}",
                        ele1.Key, ele1.Value);
          }
          Console.WriteLine();
```

```
          // Creating another dictionary
          // using Dictionary<TKey,TValue> class
      // adding key/value pairs without
          // using Add method
      Dictionary<string, string> My_dict2 =
              new Dictionary<string, string>(){
                                  {"a.1", "Dog"},
                                  {"a.2", "Cat"},
                                 {"a.3", "Pig"} };

          foreach(KeyValuePair<string, string> ele2 in My_dict2)
          {
              Console.WriteLine("{0} and {1}", ele2.Key, ele2.Value);
          }
    }
}
```

Output

```
1123 and Welcome
1124 and to
1125 and GeeksforGeeks

a.1 and Dog
a.2 and Cat
a.3 and Pig
```

Removing Elements from dictionary

In Dictionary, you are allowed to remove elements from the Dictionary. Dictionary<TKey, TValue> class provides two different methods to remove elements and the methods are:

- Clear**:** This method removes all keys and values from the Dictionary<TKey,TValue>.
- Remove**:** This method removes the value with the specified key from the Dictionary<TKey,TValue>.

Example:

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

class GFG {

        // Main Method
    static public void Main() {

        // Creating a dictionary
        // using Dictionary<TKey,TValue> class
        Dictionary<int, string> My_dict =
                    new Dictionary<int, string>();

          // Adding key/value pairs in the
          // Dictionary Using Add() method
          My_dict.Add(1123, "Welcome");
          My_dict.Add(1124, "to");
          My_dict.Add(1125, "GeeksforGeeks");

          // Before Remove() method
          foreach(KeyValuePair<int, string> ele in My_dict)
          {
              Console.WriteLine("{0} and {1}",
                          ele.Key, ele.Value);
          }
          Console.WriteLine();
```

```
          My_dict.Remove(1123);

          // After Remove() method
          foreach(KeyValuePair<int, string> ele in My_dict)
          {
              Console.WriteLine("{0} and {1}",
                          ele.Key, ele.Value);
          }
          Console.WriteLine();


          // Using Clear() method
          My_dict.Clear();

          Console.WriteLine("Total number of key/value "+
           "pairs present in My_dict:{0}", My_dict.Count);

    }
}
```

Check Availability of elements in the dictionary

In Dictionary, you can check whether the given key or value is present in the specified dictionary or not. The Dictionary<TKey, TValue> class provides two different methods for checking and the methods are:

- ContainsKey: This method is used to check whether the Dictionary<TKey,TValue> contains the specified key.
- ContainsValue: This method is used to check whether the Dictionary<TKey,TValue> contains a specific value.

Example:

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

class GFG {

        // Main Method
    static public void Main () {

                // Creating a dictionary
        // using Dictionary<TKey,TValue> class
        Dictionary<int, string> My_dict =
                    new Dictionary<int, string>();

          // Adding key/value pairs in the
          // Dictionary Using Add() method
          My_dict.Add(1123, "Welcome");
          My_dict.Add(1124, "to");
          My_dict.Add(1125, "GeeksforGeeks");

          // Using ContainsKey() method to check
          // the specified key is present or not
          if (My_dict.ContainsKey(1122)==true)
          {
              Console.WriteLine("Key is found...!!");
          }
```

```
        else
        {
             Console.WriteLine("Key is not found...!!");
        }

        // Using ContainsValue() method to check
        // the specified value is present or not
        if (My_dict.ContainsValue("GeeksforGeeks")==true)
        {
            Console.WriteLine("Value is found...!!");
        }

        else
        {
             Console.WriteLine("Value is not found...!!");
        }
    }
}
```

Output

```
Key is not found...!!
Value is found...!!
```

SETS

A **HashSet<T>** is an unordered collection of the **unique elements**. It comes under ***System.Collections.Generic*** namespace. It is used in a situation where we want to prevent duplicates from being inserted in the collection. As far as performance is concerned, it is better in comparison to the list.

Characteristics of HashSet Class:

- The HashSet<**T**> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.
- The capacity of a HashSet<**T**> object is the number of elements that the object can hold.
- A HashSet<**T**> object's capacity automatically increases as elements are added to the object.

- A HashSet<**T**> collection is not sorted and cannot contain duplicate elements.
- HashSet<**T**> provides many mathematical set operations, such as set addition (unions) and set subtraction.

Example:

```
// C# code to create a HashSet
using System;
using System.Collections.Generic;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a HashSet of odd numbers
        HashSet<int> odd = new HashSet<int>();

        // Inserting elements in HashSet
        for (int i = 0; i < 5; i++) {
            odd.Add(2 * i + 1);
        }

        // Displaying the elements in the HashSet
        foreach(int i in odd)
        {
            Console.WriteLine(i);
        }
    }
}
```

Output:

```
1
3
5
7
9
```

Properties:

Property	Description
Comparer	Gets the IEqualityComparer object that is used to determine equality for the values in the set.
Count	Gets the number of elements that are contained in a set.

Example:

```
// C# code to get the number of
// elements that are contained in HashSet
using System;
using System.Collections.Generic;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a HashSet of integers
        HashSet<int> mySet = new HashSet<int>();

        // Inserting elements in HashSet
        for (int i = 0; i < 5; i++) {
            mySet.Add(i * 2);
        }

        // To get the number of
        // elements that are contained in HashSet
        Console.WriteLine(mySet.Count);
    }
}
```

Output:

5

HashSet Methods

Method	Description
Add(T)	Adds the specified element to a set.
Clear()	Removes all elements from a HashSet object.
Contains(T)	Determines whether a HashSet object contains the specified element.
CopyTo()	Copies the elements of a HashSet collection to an array.
CreateSetComparer()	Returns an IEqualityComparer object that can be used for equality testing of a HashSet object.
Equals(Object)	Determines whether the specified object is equal to the current object.
ExceptWith(IEnumerable)	Removes all elements in the specified collection from the current HashSet object.
GetEnumerator()	Returns an enumerator that iterates through a HashSet object.
GetHashCode()	Serves as the default hash function.
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize a HashSet object.
GetType()	Gets the Type of the current instance.
IntersectWith(IEnumerable)	Modifies the current HashSet object to contain only elements that are present in that object and in the specified collection.
IsProperSubsetOf(IEnumerable)	Determines whether a HashSet object is a proper subset of the specified collection.
IsProperSupersetOf(IEnumerable)	Determines whether a HashSet object is a proper superset of the specified collection.
IsSubsetOf(IEnumerable)	Determines whether a HashSet object is a subset of the specified collection.
IsSupersetOf(IEnumerable)	Determines whether a HashSet object is a superset of the specified collection.

Remove(T)	Removes the specified element from a HashSet object.
RemoveWhere(Predicate)	Removes all elements that match the conditions defined by the specified predicate from a HashSet collection.
SetEquals(IEnumerable)	Determines whether a HashSet object and the specified collection contain the same elements.
SymmetricExceptWith(IEnumerable)	Modifies the current HashSet object to contain only elements that are present either in that object or in the specified collection, but not both.
ToString()	Returns a string that represents the current object.
TrimExcess()	Sets the capacity of a HashSet object to the actual number of elements it contains, rounded up to a nearby, implementation-specific value.
TryGetValue(T, T)	Searches the set for a given value and returns the equal value it finds, if any.

Example:

```
// C# code to Check if a HashSet is
// a subset of the specified collection
using System;
using System.Collections.Generic;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a HashSet of integers
        HashSet<int> mySet1 = new HashSet<int>();

        // Inserting elements in HashSet
        // mySet1 only contains even numbers less than
        // equal to 10
        for (int i = 1; i <= 5; i++)
            mySet1.Add(2 * i);

        // Creating a HashSet of integers
        HashSet<int> mySet2 = new HashSet<int>();

        // Inserting elements in HashSet
        // mySet2 contains all numbers from 1 to 10
        for (int i = 1; i <= 10; i++)
            mySet2.Add(i);

        // Check if a HashSet mySet1 is a subset
        // of the HashSet mySet2
        Console.WriteLine(mySet1.IsSubsetOf(mySet2));
    }
}
```

Output:

True

Example:

```
// C# code to check if a HashSet
// contains the specified element
using System;
using System.Collections.Generic;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a HashSet of strings
        HashSet<string> mySet = new HashSet<string>();

        // Inserting elements in HashSet
        mySet.Add("DS");
        mySet.Add("C++");
        mySet.Add("Java");
        mySet.Add("JavaScript");

        // Check if a HashSet contains
        // the specified element
        if (mySet.Contains("Java"))
            Console.WriteLine("Required Element is present");
        else
            Console.WriteLine("Required Element is not present");
    }
}
```

Output:

Required Element is not present

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 dequeue . This class comes under System.Collections namespace and implements *ICollection, IEnumerable, and ICloneable* interfaces.

Characteristics of Queue Class:

- Enqueue adds an element to the end of the Queue.
- Dequeue removes the oldest element from the start of the Queue.

- Peek returns the oldest element that is at the start of the Queue but does not remove it from the Queue.
- The capacity of a Queue is the number of elements the Queue can hold.
- As elements are added to a Queue, the capacity is automatically increased as required by reallocating the internal array.
- Queue accepts null as a valid value for reference types and allows duplicate elements.

Constructors

Constructor	Description
Queue()	Initializes a new instance of the Queue class that is empty, has the default initial capacity, and uses the default growth factor.
Queue(ICollection)	Initializes a new instance of the Queue class that contains elements copied from the specified collection, has the same initial capacity as the number of elements copied, and uses the default growth factor.
Queue(Int32)	Initializes a new instance of the Queue class that is empty, has the specified initial capacity, and uses the default growth factor.
Queue(Int32, Single)	Initializes a new instance of the Queue class that is empty, has the specified initial capacity, and uses the specified growth factor.

Example:

```
// C# code to create a Queue
using System;
using System.Collections;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a Queue
        Queue myQueue = new Queue();

        // Inserting the elements into the Queue
        myQueue.Enqueue("one");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("two");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("three");
```

```
        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("four");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("five");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("six");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);
    }
}
```

Output:

```
Total number of elements in the Queue are : 1
Total number of elements in the Queue are : 2
Total number of elements in the Queue are : 3
Total number of elements in the Queue are : 4
Total number of elements in the Queue are : 5
Total number of elements in the Queue are : 6
```

Properties

Property	Description
Count	Gets the number of elements contained in the Queue.
IsSynchronized	Gets a value indicating whether access to the Queue is synchronized (thread safe).
SyncRoot	Gets an object that can be used to synchronize access to the Queue.

Example:

```
// C# code to Get the number of
// elements contained in the Queue
using System;
using System.Collections;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a Queue
        Queue myQueue = new Queue();

        // Inserting the elements into the Queue
        myQueue.Enqueue("Chandigarh");
        myQueue.Enqueue("Delhi");
        myQueue.Enqueue("Noida");
        myQueue.Enqueue("Himachal");
        myQueue.Enqueue("Punjab");
        myQueue.Enqueue("Jammu");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);
    }
}
```

Output:

```
Total number of elements in the Queue are : 6
```

Methods

Method	Description
Clear()	Removes all objects from the Queue.
Clone()	Creates a shallow copy of the Queue.
Contains(Object)	Determines whether an element is in the Queue.
CopyTo(Array, Int32)	Copies the Queue elements to an existing one-dimensional Array, starting at the specified array index.
Dequeue()	Removes and returns the object at the beginning of the Queue.
Enqueue(Object)	Adds an object to the end of the Queue.
Equals(Object)	Determines whether the specified object is equal to the current object.
GetEnumerator()	Returns an enumerator that iterates through the Queue.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object.
Peek()	Returns the object at the beginning of the Queue without removing it.
Synchronized(Queue)	Returns a new Queue that wraps the original queue, and is thread safe.
ToArray()	Copies the Queue elements to a new array.
ToString()	Returns a string that represents the current object.
TrimToSize()	Sets the capacity to the actual number of elements in the Queue.

Example:

```
// C# code to Check if a Queue
// contains an element
using System;
using System.Collections;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a Queue
        Queue myQueue = new Queue();

        // Inserting the elements into the Queue
        myQueue.Enqueue(5);
        myQueue.Enqueue(10);
        myQueue.Enqueue(15);
        myQueue.Enqueue(20);
        myQueue.Enqueue(25);

        // Checking whether the element is
        // present in the Queue or not
        // The function returns True if the
        // element is present in the Queue, else
        // returns False
        Console.WriteLine(myQueue.Contains(7));
    }
}
```

Output:

```
False
```

Example 2:

```
// C# code to Convert Queue to array
using System;
using System.Collections;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a Queue
        Queue myQueue = new Queue();

        // Inserting the elements into the Queue
        myQueue.Enqueue("Geeks");
        myQueue.Enqueue("Geeks Classes");
        myQueue.Enqueue("Noida");
        myQueue.Enqueue("Data Structures");
        myQueue.Enqueue("GeeksforGeeks");

        // Converting the Queue
        // into object array
        Object[] arr = myQueue.ToArray();

        // Displaying the elements in array
        foreach(Object obj in arr)
        {
            Console.WriteLine(obj);
        }
    }
}
```

Output:

```
Geeks
Geeks Classes
Noida
Data Structures
GeeksforGeeks
```

STACK

A Stack represents a last-in, first-out collection of objects. It is used when you need last-in, first-out access to items. It is both a generic and non-generic type of collection. The generic stack is defined in *System.Collections.Generic* namespace whereas non-generic stack is defined under *System.Collections* namespace, here we will discuss non-generic type stack. A stack is used to create a dynamic collection that grows, according to the need of your program. In a stack, you can store elements of the same type or different types.

Important Points:

- The Stack class implements the *IEnumerable*, *ICollection*, and *ICloneable* interfaces.
- When you add an item in the list, it is called *pushing* the element.
- When you remove it, it is called *popping* the element.
- The capacity of a Stack is the number of elements the Stack can hold. As elements are added to a Stack, the capacity is automatically increased as required through reallocation.
- In Stack, you are allowed to store duplicate elements.
- A Stack accepts null as a valid value for reference types.

How to create a Stack?

Stack class has *three* constructors which are used to create a stack which is as follows:

- **Stack():** This constructor is used to create an instance of the Stack class which is empty and having the default initial capacity.
- **Stack(ICollection):** This constructor is used to create an instance of the Stack class which contains elements copied from the specified collection, and has the same initial capacity as the number of elements copied.
- **Stack(Int32):** This constructor is used to create an instance of the Stack class which is empty and having specified initial capacity or the default initial capacity, whichever is greater.

Let's see how to create a stack using Stack() constructor:

Step 1: Include *System.Collections* namespace in your program with the help of using keywords.

```
using System.Collections;
```

Step 2: Create a stack using Stack class as shown below:

```
Stack stack_name = new Stack();
```

Step 3: If you want to add elements in your stack, then use *Push()* method to add elements in your stack. As shown in the below example.

Example:

```
// C# program to illustrate how to
// create a stack
using System;
using System.Collections;

class GFG {

    // Main Method
    static public void Main()
    {

        // Create a stack
        // Using Stack class
        Stack my_stack = new Stack();

        // Adding elements in the Stack
        // Using Push method
        my_stack.Push("Geeks");
        my_stack.Push("geeksforgeeks");
        my_stack.Push('G');
        my_stack.Push(null);
        my_stack.Push(1234);
        my_stack.Push(490.98);

        // Accessing the elements
        // of my_stack Stack
        // Using foreach loop
        foreach(var elem in my_stack)
        {
            Console.WriteLine(elem);
        }
    }
}
```

Output

```
490.98
1234

G
geeksforgeeks
Geeks
```

How to remove elements from the Stack?

In Stack, you are allowed to remove elements from the stack. The Stack class provides two different methods to remove elements and the methods are:

- Clear: This method is used to remove all the objects from the stack.
- Pop: This method removes the beginning element of the stack.

Example:

```
// remove elements from the stack
using System;
using System.Collections;

class GFG {

    // Main Method
    static public void Main()
    {

        // Create a stack
        // Using Stack class
        Stack my_stack = new Stack();

        // Adding elements in the Stack
        // Using Push method
        my_stack.Push("Geeks");
        my_stack.Push("geeksforgeeks");
        my_stack.Push("geeks23");
        my_stack.Push("GeeksforGeeks");

        Console.WriteLine("Total elements present in"+
                    " my_stack: {0}", my_stack.Count);

        my_stack.Pop();

        // After Pop method
        Console.WriteLine("Total elements present in "+
                      "my_stack: {0}", my_stack.Count);


        // Remove all the elements
        // from the stack
        my_stack.Clear();

        // After Pop method
        Console.WriteLine("Total elements present in "+
                      "my_stack: {0}", my_stack.Count);

    }
}
```

Output:

```
Total elements present in my_stack: 4
Total elements present in my_stack: 3
Total elements present in my_stack: 0
```

How to get the topmost element of the Stack?

In Stack, you can easily find the topmost element of the stack by using the following methods provided by the Stack class:

- **Pop:** This method returns the object at the beginning of the stack with modification means this method removes the topmost element of the stack.
- **Peek:** This method returns the object at the beginning of the stack without removing it.

Example:

```
// C# program to illustrate how to
// get topmost elements of the stack
using System;
using System.Collections;

class GFG {

    // Main Method
    static public void Main()
    {

        // Create a stack
        // Using Stack class
        Stack my_stack = new Stack();

        // Adding elements in the Stack
        // Using Push method
        my_stack.Push("Geeks");
        my_stack.Push("geeksforgeeks");
        my_stack.Push("geeks23");
        my_stack.Push("GeeksforGeeks");

        Console.WriteLine("Total elements present in"+
                    " my_stack: {0}",my_stack.Count);

        // Obtain the topmost element
        // of my_stack Using Pop method
        Console.WriteLine("Topmost element of my_stack"
                        + " is: {0}",my_stack.Pop());

        Console.WriteLine("Total elements present in"+
                    " my_stack: {0}", my_stack.Count);
```

```
        // Obtain the topmost element
        // of my_stack Using Peek method
        Console.WriteLine("Topmost element of my_stack "+
                            "is: {0}",my_stack.Peek());


        Console.WriteLine("Total elements present "+
                 "in my_stack: {0}",my_stack.Count);

    }
}
```

Output:

```
Total elements present in my_stack: 4
Topmost element of my_stack is: GeeksforGeeks
Total elements present in my_stack: 3
Topmost element of my_stack is: geeks23
Total elements present in my_stack: 3
```

How to check the availability of elements in the stack?

In a stack, you can check whether the given element is present or not using the Contains() method. Or in other words, if you want to search an element in the given stack use Contains() method. This method returns true if the element is present in the stack. Otherwise, return false.

Note: The Contains() method takes O(n) time to check if the element exists in the stack. This should be taken into consideration while using this method.

Example:

```
using System;
using System.Collections;

class GFG {

    // Main  Method
    static public void Main()
    {

        // Create a stack
        // Using Stack class
        Stack my_stack = new Stack();

        // Adding elements in the Stack
        // Using Push method
        my_stack.Push("Geeks");
        my_stack.Push("geeksforgeeks");
        my_stack.Push("geeks23");
        my_stack.Push("GeeksforGeeks");

        // Checking if the element is
        // present in the Stack or not
        if (my_stack.Contains("GeeksforGeeks") == true)
        {
            Console.WriteLine("Element is found...!!");
        }

        else
        {
            Console.WriteLine("Element is not found...!!");
        }
    }
}
```

Output:

```
Element is found...!!
```

Generic Stack Vs Non-Generic Stack

Generic Stack	Non-Generic Stack
Generic stack is defined under System.Collections.Generic namespace.	Non-Generic stack is defined under System.Collections namespace.
Generic stack can only store same type of elements.	Non-Generic stack can store same type or different types of elements.
There is a need to define the type of the elements in the stack.	There is no need to define the type of the elements in the stack.
It is type-safe.	It is not type-safe.