ASSIGNMENT

DICTIONARY

In C#, Dictionary is a generic collection which is generally 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 means the size of the dictionary is grows according to the need.

Important Points:

The Dictionary class implements the
IDictionary<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, 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 same types of elements.
The capacity of a Dictionary is the number of elements that Dictionary can hold.
How to create the 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:

```
// C# program to illustrate how
// to create a dictionary
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
```

How to remove elements from the 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:

```
// C# program to illustrate how
// remove key/value pairs from
// the dictionary
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();

        // Using Remove() method
        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);

    }
}
```

Output:

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

1124 and to
1125 and GeeksforGeeks

Total number of key/value pairs present in My_dict:0
```

How to check the availability of elements in the Dictionary?

In Dictionary, you can check whether the given key or value 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:

```
// C# program to illustrate how
// to  check the given key or
// value present in the dictionary
// or not
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...!!
```

STACK

`Stack` is a special type of collection that stores elements in LIFO style (Last In First Out). C# includes the generic `Stack<T>` and non-generic `Stack` collection classes. It is recommended to use the generic `Stack<T>` collection.

Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.

Stack<T> Characteristics

- `Stack<T>` is Last In First Out collection.
- It comes under `System.Collection.Generic` namespace.
- `Stack<T>` can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the `Push()` method. Cannot use collection-initializer syntax.
- Elements can be retrieved using the `Pop()` and the `Peek()` methods. It does not support an indexer.

Creating a Stack

You can create an object of the `Stack<T>` by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the `Stack<T>` using the `Push()` method. Stack allows null (for reference types) and duplicate values.

Example: Create and Add Elements in Stack

```
Stack<int> myStack = new Stack<int>(); myStack.Push(1); myStack.Push(2); myStack.Push(3); myStack.Push(4); foreach (var item in myStack) Console.Write(item + ","); //prints 4,3,2,1,
```

Try it

You can also create a Stack from an array, as shown below.

Example: Create and Add Elements in Stack

```
int[] arr = new int[]{ 1, 2, 3, 4}; Stack<int> myStack = new Stack<int>(arr); foreach (var item in myStack) Console.Write(item + ","); //prints 4,3,2,1,
```

Try it

Stack<T> Properties and Methods:

Property	Usage
Count	Returns the total count of elements in the Stack.

Method	Usage
Push(T)	Inserts an item at the top of the stack.
Peek()	Returns the top item from the stack.
Pop()	Removes and returns items from the top of the stack.
Contains(T)	Checks whether an item exists in the stack or not.
Clear()	Removes all items from the stack.

Pop()

The `Pop()` method returns the last element and removes it from a stack. If a stack is empty, then it will throw the `InvalidOperationException`. So, always check for the number of elements in a stack before calling the `Pop()` method.

Example: Access Stack using Pop()

```
Stack<int> myStack = new Stack<int>(); myStack.Push(1);
myStack.Push(2); myStack.Push(3); myStack.Push(4);
Console.Write("Number of elements in Stack: {0}",
myStack.Count); while (myStack.Count > 0)
Console.Write(myStack.Pop() + ","); Console.Write("Number of
elements in Stack: {0}", myStack.Count);
```

Try it

Output:

```
Number of elements in Stack: 4
4,3,2,1,
Number of elements in Stack: 0
```

Peek()

The `Peek()` method returns the lastly added value from the stack but does not remove it. Calling the `Peek()` method on an empty stack will throw the `InvalidOperationException`. So, always check for elements in the stack before retrieving elements using the `Peek()` method.

Example: Retrieve Elements usign Peek()

```
Stack<int> myStack = new Stack<int>(); myStack.Push(1);
myStack.Push(2); myStack.Push(3); myStack.Push(4);
Console.Write("Number of elements in Stack: {0}",
myStack.Count);// prints 4 if(myStack.Count > 0){
Console.WriteLine(myStack.Peek()); // prints 4
Console.WriteLine(myStack.Peek()); // prints 4 }
Console.Write("Number of elements in Stack: {0}",
myStack.Count);// prints 4
```

Try it

Contains()

The `Contains()` method checks whether the specified element exists in a Stack collection or not. It returns true if it exists, otherwise false.

Example: Contains()

```
Stack<int> myStack = new Stack<int>(); myStack.Push(1); myStack.Push(2); myStack.Push(3); myStack.Push(4); myStack.Contains(2); // returns true myStack.Contains(10); // returns false
```

QUEUE

Represents a first-in, first-out collection of objects.

C#

Copy

```
public class Queue<T> : System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.ICollection
```

Type Parameters

T

Specifies the type of elements in the queue.

Inheritance

Object Queue<T>

Implements

IEnumerable<T> IReadOnlyCollection<T> ICollection IEnumerable

Examples

The following code example demonstrates several methods of the Queue<T> generic class. The code example creates a queue of strings with default capacity and uses the Enqueue method to queue five strings. The elements of the queue are enumerated, which does not change the state of the queue. The Dequeue method is used to dequeue the first string. The Peek method is used to look at the next item in the queue, and then the Dequeue method is used to dequeue it.

The ToArray method is used to create an array and copy the queue elements to it, then the array is passed to the Queue<T> constructor that takes IEnumerable<T>, creating a copy of the queue. The elements of the copy are displayed.

An array twice the size of the queue is created, and the CopyTo method is used to copy the array elements beginning at the middle of the array. The Queue<T> constructor is used again to create a second copy of the queue containing three null elements at the beginning.

The Contains method is used to show that the string "four" is in the first copy of the queue, after which the Clear method clears the copy and the Count property shows that the queue is empty.

C#

Copy

Run

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

class Example
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");

        // A queue can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
        Console.WriteLine("Peek at next item to dequeue: {0}",
            numbers.Peek());
        Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());

        // Create a copy of the queue, using the ToArray method and
the
```

```
        // constructor that accepts an IEnumerable<T>.
        Queue<string> queueCopy = new
Queue<string>(numbers.ToArray());

        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in queueCopy )
        {
            Console.WriteLine(number);
        }

        // Create an array twice the size of the queue and copy the
        // elements of the queue, starting at the middle of the
        // array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);

        // Create a second queue, using the constructor that accepts
an
        // IEnumerable(Of T).
        Queue<string> queueCopy2 = new Queue<string>(array2);

        Console.WriteLine("\nContents of the second copy, with
duplicates and nulls:");
        foreach( string number in queueCopy2 )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
            queueCopy.Contains("four"));

        Console.WriteLine("\nqueueCopy.Clear()");
        queueCopy.Clear();
        Console.WriteLine("\nqueueCopy.Count = {0}",
queueCopy.Count);
    }
}

/* This code example produces the following output:

one
two
three
four
five
```

```
Dequeuing 'one'
Peek at next item to dequeue: two
Dequeuing 'two'

Contents of the first copy:
three
four
five

Contents of the second copy, with duplicates and nulls:



three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
 */
```

Remarks

This class implements a generic queue as a circular array. Objects stored in a Queue<T> are inserted at one end and removed from the other. Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue<T> if you need to access the information in the same order that it is stored in the collection. Use Stack<T> if you need to access the information in reverse order. Use ConcurrentQueue<T> or ConcurrentStack<T> if you need to access the collection from multiple threads concurrently.

Three main operations can be performed on a Queue<T> and its elements:

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

- Peek peek returns the oldest element that is at the start of the Queue<T> but does not remove it from the Queue<T>.

The capacity of a Queue<T> is the number of elements the Queue<T> can hold. As elements are added to a Queue<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

Queue<T> accepts `null` as a valid value for reference types and allows duplicate elements.

Constructors

Queue<T>()	Initializes a new instance of the Queue<T> class that is empty and has the default initial capacity.
Queue<T>(IEnumerable<T>)	Initializes a new instance of the Queue<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
Queue<T>(Int32)	Initializes a new instance of the Queue<T> class that is empty and has the specified initial capacity.

Properties

Count	Gets the number of elements contained in the Queue<T>.

Methods

Clear()	Removes all objects from the Queue<T>.
Contains(T)	Determines whether an element is in the Queue<T>.
CopyTo(T[], Int32)	Copies the Queue<T> 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<T>.
Enqueue(T)	Adds an object to the end of the Queue<T>.
EnsureCapacity(Int32)	Ensures that the capacity of this queue is at least the specified `capacity`. If the current capacity is less than `capacity`, it is successively increased to twice the current capacity until it is at least the specified `capacity`.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Queue<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Peek()	Returns the object at the beginning of the Queue<T> without removing it.
ToArray()	Copies the Queue<T> elements to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of elements in the Queue<T>, if that number is less than 90 percent of current capacity.

TryDequeue(T)	Removes the object at the beginning of the Queue<T>, and copies it to the `result` parameter.
TryPeek(T)	Returns a value that indicates whether there is an object at the beginning of the Queue<T>, and if one is present, copies it to the `result` parameter. The object is not removed from the Queue<T>.