**Arrays in C#**

**Arrays in C# with Examples**

In this article, I am going to discuss the **Arrays in C#** with examples. It is one of the important concepts in programming languages. Arrays are there from our traditional programming languages such as C, C++ and are also available in C#. As part of this article, we are going to discuss the following pointers.

1. **Why do we need arrays in programming?**
2. **What is an Array in C#?**
3. **Types of Arrays in C#.**
4. **Understanding the memory representation of the array in C#.**
5. **One Dimensional Array in C# with Examples**
6. **What is the difference between for loop and for each loop in C# to access array values?**
7. **What is the Array class in C#?**
8. **Understanding the Array class methods and properties.**
9. **What is Implicitly Type Array in C#?**

**Why do we need Arrays in programming?**

As we know a primitive type variable such as int, double can hold only a single value at any given point in time. For example, **int no = 10;**. Here the variable **“no”** holds a value of **10**. As per your business requirement, if you want to store 100 integer values, then you need to create 100 integer variables which is not a good programming approach as it will take lots of time as well as your code becomes bigger. So to overcome the above problems, Arrays in C# are introduced.

**What is an Array in C#?**

In simple words, we can define an array as a collection of similar types of values that are stored in sequential order i.e. they are stored in a contiguous memory location.

**Types of Arrays in C#:**

C# supports 2 types of arrays. They are as follows:

1. **Single dimensional array**
2. **Multi-dimensional array**

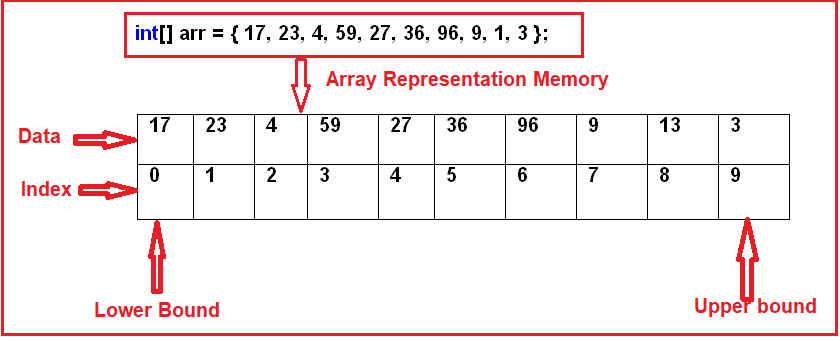
In the Single dimensional array, the data is arranged in the form of a row whereas in the Multi-dimensional arrays in C# the data is arranged in the form of rows and columns. Again the multi-dimensional arrays are of two types

1. **Jagged array**: Whose rows and columns are not equal
2. **Rectangular array**: Whose rows and columns are equal

We can access the values of an array using the index positions whereas the array index starts from 0 which means the first item of an array will be stored at the 0th position and the position of the last item of an array will be the total number of the item – 1.

**Memory Representation of Arrays in C#:**

Please have a look at the following diagram:



As you can see in the above diagram, we have an integer array with 10 elements. The array index is starting from 0, which stores the first element of the array. As the array contains 10 elements, so the last index position will be 9. The Array values or elements are stored sequentially in the memory i.e. contiguous memory location and this is the reason why it performs faster.

In C#, the arrays can be declared as fixed-length or dynamic. The Fixed length array means we can store a fixed number of elements while in the case of the dynamic array, the size of the array automatically increases as we add new items into the array.

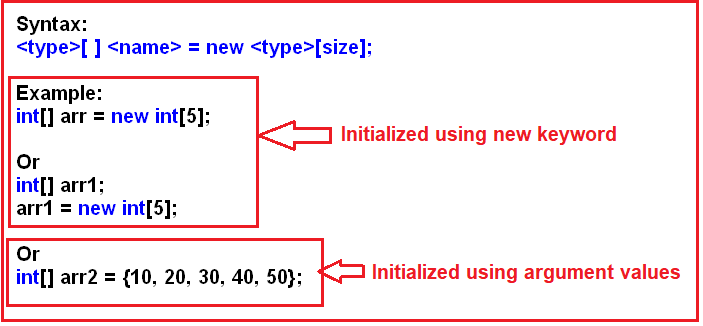
**Note:**The Arrays in C# are reference types that are derived from the System.Array class.

**Can we use a for each loop to iterate on arrays in C#?**

Yes. Since the arrays in C# are derived from the **System.Array** class which implements the **IEnumerable**, so we can use the for-eachloop to iterate on arrays in C#.

**One Dimensional Array in C# with Examples:**

The array which stores the data in the form of rows in a sequential order is called a one-dimensional array in C#. The syntax for creating a one-dimensional array in C# is given below.



As you can see in the above image, we can initialize an array in C# either by using the **new** keyword or using the **argument** values.

**One-dimensional Array Example in C#.**

In the below example, first, we create an array with size 6. To check what default values an array in c# store, without initializing the array, we are printing the values on the console using a for loop. Then again, using a for loop we are assigning the elements to the array. Finally, we are accessing the array elements and printing the values on the console using a for each loop.

**namespace** *ArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating an array with size 6

**int[]** arr = new **int[**6**]**;

//accessing array values using loop

//Here it will display the default values

//as we are not assigning any values

**for** **(int** i = 0; i **<** 6; i++**)**

**{**

Console.Write**(**arr**[**i**]** + " "**)**;

**}**

Console.WriteLine**()**;

**int** a = 0;

//Here we are assigning values to array using for loop

**for** **(int** i = 0; i **<** 6; i++**)**

**{**

a += 10;

arr**[**i**]** = a;

**}**

//accessing array values using foreach loop

**foreach** **(int** i in arr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Output: As you can see in the below output, the default values 0, will store for integer type array.

Arrays in C#

In this example, we have used a special loop called for each loop to access the array elements in C#. Let us first understand what this for each loop is and then we will see the difference between for and for each loop in C#.

**For each loop in C#:**

This for each loop is specially designed in C# for accessing the values from a collection like an array. When we use a for-each loop for accessing the values of an array or collection, we only require to hand over the array or collection to the loop which does not require any initialization, condition, or iteration. The loop itself starts its execution by providing access to each and every element present in the array or collection starting from the first up to the last element in sequential order.

**What is the difference between for loop and for each loop in C# to access array values?**

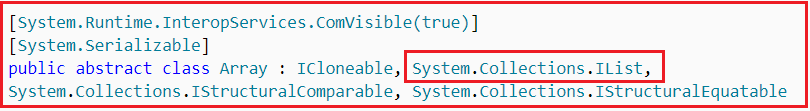
In the case of for loop in C#, the loop variable refers to the index of an array whereas, in the case of a for-each loop, the loop variable refers to the values of the array.

Irrespective of the values stored in the array, the loop variable must be of type **int** in case of for loop. The reason for this is, here the loop variable is referring to the index position of the array.  Coming to the for-each loop, the data type of the loop variable must be the same as the type of the values stored in the array. For example, if you have a string array then the loop variable must be of type **string** in case of the for-each loop in C#.

The most important point that you need to keep in mind is that the for loop in C# can be used both for accessing values from an array as well as assigning values to an array whereas the for-each loop in C# can only be used for accessing the values from an array but not for assigning values into an array.

**What is the Array class in C#?**

The **Array** class is a predefined class that is defined inside the **System** namespaces. This class is working as the base class for all the arrays in C#. The **Array** class provides a set of members (methods and properties) to work with the arrays such as creating, manipulating, searching, reversing, and sorting the elements of an array, etc. The definition of the Array class in C# is gen below.



The Array class in C# is not a part of the **System.Collections** namespace. It is a part of the **System** namespace. But still, we considered it as a collection because it is Implements the **IList** interface. The Array class provides the following methods and properties:

1. **Sort(<array>):**Sorting the array elements
2. **Reverse (<array>):**Reversing the array elements
3. **Copy (src, dest, n):**Copying some of the elements or all elements from the old array to the new array
4. **GetLength(int):**A 32-bit integer that represents the number of elements in the specified dimension.
5. **Length:**It Returns the total number of elements in all the dimensions of the Array; zero if there are no elements in the array.

Example: Array class Methods and Properties in C#

Let’s see an example for understanding the Method and Properties of the Array class in C#.

**using** *System;*

**namespace** *ArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Declaring an array

**int[]** arr = **{** 17, 23, 4, 59, 27, 36, 96, 9, 1, 3 **}**;

//Printing the array elements

Console.WriteLine**(**"The Array Contains the Below Elements:"**)**;

**for** **(int** i = 0; i **<** arr.Length; i++**)**

**{**

Console.Write**(**arr**[**i**]** + " "**)**;

**}**

Console.WriteLine**()**;

//Sorting the array elements

Array.Sort**(**arr**)**;

//Printing the array elements after sorting

Console.WriteLine**(**"The Array Elements After Sorting:"**)**;

**foreach** **(int** i in arr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

Console.WriteLine**()**;

//Reversing the array elements

Array.Reverse**(**arr**)**;

//Printing the array elements in reverse order

Console.WriteLine**(**"The Array Elements in the Reverse Order :"**)**;

**foreach** **(int** i in arr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

Console.WriteLine**()**;

//Creating a new array

**int[]** brr = new **int[**10**]**;

//Copying some of the elements from old array to new array

Console.WriteLine**(**"The new array elements:"**)**;

Array.Copy**(**arr, brr, 5**)**;

//We declare the array with size 10. we copy only 5 elements. so the rest

//5 elements will take the default value. In this array, it will take 0

**foreach** **(int** i in brr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"brr Array Length :" + brr.Length**)**;

Console.WriteLine**(**"brr Array Length :" + brr.GetLength**(**0**))**;

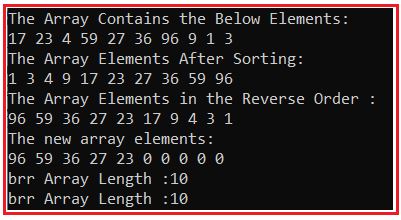
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Understanding the Implicitly Typed Arrays in C#:**

When we declare an array by using the “var” keyword then such types of arrays are called implicitly typed arrays in C#.  
Example: **var arr = new[] {10, 20, 30 , 40, 50};**  
Let us see an example for understanding the implicitly typed array in C#.

**namespace** *ArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var arr = new**[]** **{** 17, 23, 4, 59, 27, 36, 96, 9, 1, 3 **}**;

**for** **(int** i = 0; i **<** arr.Length; i++**)**

**{**

Console.Write**(**arr**[**i**]** + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Understanding Implicilt Typed Array in C#

**2d Array in C#**

**2d Array in C# with Examples**

In this article, I am going to discuss the **2d Array in C#** with Examples. Please read our previous article before proceeding to this article where we discussed [**one-dimensional Arrays in C#**](https://dotnettutorials.net/lesson/arrays-csharp/) with examples.  As part of this article, we are going to discuss the following pointers which are related to the Two-Dimensional Array in C#.

1. **What is a Two-Dimensional Array in C#?**
2. **Understanding the rectangular and jagged array in C# with examples.**

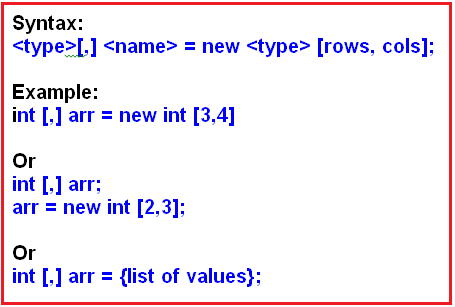
**What is a Two-Dimensional Array in C#?**

The arrays which store the elements in the form of rows and columns are called Two-Dimensional Array in C#. The two-dimensional array which is also called multidimensional array is of two types in C#. They are as follows

1. **Rectangular array**: The array whose rows and columns are equal are called a rectangular array
2. **Jagged array**: The array whose rows and columns are not equal are called a jagged array

**Rectangular 2d Array in C#:**

Let us first understand the syntax of the Two-Dimensional Array in C#. Please have a look at the following diagram.



**Let us see an example for a better understanding of the rectangular array in C#.**

**namespace** *TwoDimensionalArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[**,**]** arr = new **int[**4, 5**]**;

**int** a = 0;

//printing the values of 2d array using foreach loop

//It will print the default values as we are not assigning

//any values to the array

**foreach** **(int** i in arr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

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

//assigning values to the array by using nested for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr.GetLength**(**1**)**; j++**)**

**{**

a += 5;

arr**[**i, j**]** = a;

**}**

**}**

//printing the values of array by using nested for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr.GetLength**(**1**)**; j++**)**

**{**

Console.Write**(**arr**[**i, j**]** + " "**)**;

**}**

**}**

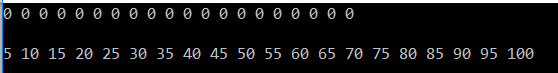
Console.ReadKey**()**;

**}**

**}**

**}**

**Output**:



In the above example, we assigned the two-dimensional array element using nested for loop. It is also possible that we can assign the values to a two-dimensional array in C# at the time of its declaration:

**Assigning values to two-dimensional array in C# at the time of declaration:**

**namespace** *TwoDimensionalArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Assigning the array elements at the time of declaration

**int[**,**]** arr = **{{**11,12,13,14**}**,

**{**21,22,23,24**}**,

**{**31,32,33,34**}}**;

//printing values of array using for each loop

**foreach** **(int** i in arr**)**

**{**

Console.Write**(**i + " "**)**;

**}**

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

//printing the values of array using nested for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr.GetLength**(**1**)**; j++**)**

**{**

Console.Write**(**arr**[**i, j**]** + " "**)**;

**}**

**}**

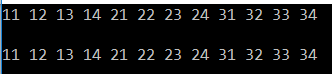
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Jagged Array in C#:**

These are also two-dimensional arrays that will also store the data in the forms of rows and columns. But here in the jagged array, the column size will differ from row to row. That means if the first row contains 5 columns then the second row may contain 4 columns while the third row may contain 10 columns. So the point that you need to remember is if the column size varies from row to row then it is a jagged array. If the column size remains the same for all the rows then it is a rectangular two-dimensional array.

The jagged array in C# is also called the array of arrays. This is because in the case of the jagged array each row is a single dimensional array. So a combination of multiple single-dimensional arrays with different column sizes form a jagged array in C#.

**Syntax:  <type> [][] <name> = new <type> [rows][];**

**Example:**

**int** **[][]** arr = new **int[**3**][]**;

//Or

**int** **[][]** arr = **{**list of values**}**;

To declare a jagged array in C#, at the time of its declaration, you only need to specify the number of rows that you want in the array. for example

**int [][] arr = new int[4][];**

In the above array declaration, we are specifying that we want four rows in the array. Once you specify the number of rows that you want in the array, then you need to initialize each row with the number of columns by using a single dimensional array as shown below.

**arr[0] = new int[5]; // we want five columns in the first row**  
**arr[1] = new int[6]; // we want six columns in the first row**  
**arr[2] = new int[4]; // we want four columns in the first row**  
**arr[3] = new int[5]; // we want five columns in the first row**

**Example of the jagged array in C#:**

**namespace** *TwoDimensionalArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating an jagged array with four rows

**int[][]** arr = new **int[**4**][]**;

//Initializing each row with different column size

// Uisng one dimensional array

arr**[**0**]** = new **int[**5**]**;

arr**[**1**]** = new **int[**6**]**;

arr**[**2**]** = new **int[**4**]**;

arr**[**3**]** = new **int[**5**]**;

//printing the values of jagged array using nested for loop

//It will print the default values as we are assigning any

//values to the array

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr**[**i**]**.Length; j++**)**

**{**

Console.Write**(**arr**[**i**][**j**]** + " "**)**;

**}**

**}**

Console.WriteLine**()**;

//assigning values to the jagged array by using nested for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr**[**i**]**.Length; j++**)**

**{**

arr**[**i**][**j**]** = j++;

**}**

**}**

//print values the values of jagged array by using foreach loop within for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**foreach** **(int** x in arr**[**i**])**

**{**

Console.Write**(**x + " "**)**;

**}**

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

When we run the application, it will give us the following output:

Jagged Array in C#

In the above example, we are assigning the elements of the jagged array by using nested for loop. It is also possible to assign the values of the jagged array at the time of its declaration.

**Assigning jagged array at the time of its declaration:**

**namespace** *TwoDimensionalArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

// Assigning the values of the jagged array

// at the time of its declaration

**int[][]** arr = **{**

new **int[**4**]{**11,12,13,14**}**,

new **int[**5**]{**21,22,23,24,25**}**,

new **int[**3**]{**31,32,33**}**

**}**;

//printing the values of jagged array by using nested for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** arr**[**i**]**.Length; j++**)**

**{**

Console.Write**(**arr**[**i**][**j**]** + " "**)**;

**}**

**}**

Console.WriteLine**()**;

//print the values of jagged array by using foreach loop within for loop

**for** **(int** i = 0; i **<** arr.GetLength**(**0**)**; i++**)**

**{**

**foreach** **(int** x in arr**[**i**])**

**{**

Console.Write**(**x + " "**)**;

**}**

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Assigning jagged array at the time of its declaration

**Advantages and Disadvantages of Arrays in C#**

**Advantages and Disadvantages of Arrays in C#**

In this article, I am going to discuss the **Advantages and Disadvantages of Arrays in C#** with examples. This is one of the most frequently asked C# interview questions. Before proceeding to this article, I strongly recommended you read the following two articles.

[**Arrays in C#**](https://dotnettutorials.net/lesson/arrays-csharp/) – Here we discussed the basics of the array along with a one-dimensional array with examples.

[**Two-Dimensional Arrays in C#**](https://dotnettutorials.net/lesson/two-dimensional-array-in-csharp/) – Here we discussed the two-dimensional arrays along with jagged array in C# with examples.

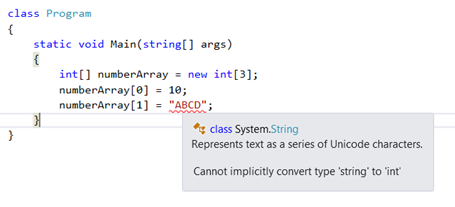
**Advantages of using an Array in C#:**

The advantages of using an array in C# are as follows:

1. It is used to represent similar types of multiple data items using a single name.
2. We can use arrays to implement other data structures such as linked lists, trees, graphs, stacks, queues, etc.
3. The two-dimensional arrays in C# are used to represent matrices.
4. The Arrays in C# are strongly typed. That means they are used to store similar types of multiple data items using a single name. As the arrays are strongly typed so we are getting two advantages. First, the performance of the application will be much better because boxing and unboxing will not happen. Secondly, runtime errors will be prevented because of a type mismatch. In this case, at compile time it will give you the error if there is a type mismatch.

**Example:**

In the following example, we create an integer array with the name numberArray. As it is an integer array so we can store only the integer values within the array. As you can see when we try to store a string value within the array, immediately it gives us a compiler error saying “**cannot implicitly convert a string to an integer**“. This is the reason why we call the arrays in C# are strongly typed.



You can get the type mismatches and runtime errors with collection classes like **ArrayList**, **Queue**, **Stack**, etc which are present in the **System.Collections** namespace. We will discuss collections in detail in our next article. But in this article, to make you understand the type mismatch let create an example using the ArrayList which is a collection in C#.

**Type Mismatch Example using ArrayList Collection in C#:**

In the following example, we create an array using the ArrayList collection with the name numberArray. The collection classes that are present in the System.Collections namespace such as **ArrayList** is loosely typed. the Loosely typed means, you can store any type of data into that collection. The **ArrayList** in C# is operated on object data type, which makes the ArrayList loosely typed. So you will not get any compile-time error, but when you run the application, you will get a runtime error.

**namespace** *ArayDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ArrayList numberArray = new ArrayList**()**;

numberArray.Add**(**10**)**;

numberArray.Add**(**200**)**;

//No CompileTime Error

numberArray.Add**(**"Pranaya"**)**;

**foreach(int** no in numberArray**)**

**{**

//We Get Runtime Error

Console.WriteLine**(**no**)**;

**}**

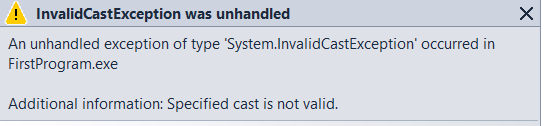
Console.ReadKey**()**;

**}**

**}**

**}**

**When you run the application, at runtime you will get the following exception.**



**Disadvantages of using Arrays in C#:**

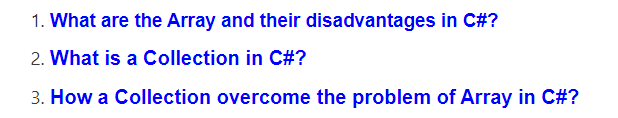
1. The array size is fixed. So, we should know in advance how many elements are going to be stored in the array. Once the array is created, then we can never increase the size of an array. If you want then we can do it manually by creating a new array and copying the old array elements into the new array.
2. As the array size is fixed, if we allocate more memory than the requirement then the extra memory will be wasted. On the other hand, if we allocate less memory than the requirement, then it will create the problem.
3. We can never insert an element into the middle of an array. It is also not possible to delete or remove elements from the middle of an array.

To overcome the above problems Collections are introduced in C#.  
In the article, we are going to start [**Collections in C#**](https://dotnettutorials.net/lesson/collections-csharp/). Here, in this article, I try to explain the **Advantages and Disadvantages of Arrays in C#**with examples. I hope this article will help you with your needs. I would like to have your feedback. Please post your feedback, question, or comments about this article.

**Collections in C#**

**Introduction to Collections in C#**

In this article, I am going to give you a brief **Introduction to Collections in C#**. Please read our previous article where we discussed the [**advantages and disadvantages of arrays in C#**](https://dotnettutorials.net/lesson/advantages-and-disadvantages-of-arrays-csharp/) with examples. As part of this article, we are going to discuss the following pointers in detail.



So, let’s first understand what is the problem with the traditional array in C#, and then we will discuss how to overcome the problems using collections in C#.

**What are the Array and their disadvantages in C#?**

In simple words, we can say that the Arrays in C# are the simple data structure that is used to store similar types of data items in sequential order. Although the arrays in c# are commonly used, they have some limitations.

For example, you need to specify the array’s size while creating the array. If at execution time, you want to modify it that means if you want to increase or decrease the size of an array, then you need to do it manually by creating a new array or by using the Array class’s Resize method, which internally creates a new array and copies the existing array element into the new array.

**Following are the limitations of Array in C#:**

1. The array size is fixed. Once the array is created we can never increase the size of an array. If we want then we can do it manually by creating a new array and copy the old array elements into the new array or by using the Array class Resize method which will do the same thing means to create a new array and copy the old array elements into the new array and then destroy the old array.
2. We can never insert an element into the middle of an array
3. Deleting or removing elements from the middle of the array.

To overcome the above problems, the Collections are introduced in C# 1.0.

**What is a Collection in C#?**

The **Collections in C#** are a set of predefined classes that are present in the **System.Collections** namespace that provides greater capabilities than the traditional arrays. The collections in C# are reusable, more powerful, more efficient and most importantly they have been designed and tested to ensure quality and performance.

So in simple words, we can say a **Collection in C#** is a **dynamic array.** That means the collections in C# have the capability of storing multiple values but with the following features.

1. Size can be increased dynamically.
2. We can insert an element into the middle of a collection.
3. It also provides the facility to remove or delete elements from the middle of a collection.

The collection which is from .Net framework 1.0 is called simply **collections** or **Non-Generic collections in C#**. These collection classes are present inside the **System.Collections** namespace. The example includes are **Stack, Queue, LinkedList, SortedList, ArrayList, HashTable,** etc.

**Auto-Resizing of collections:**

The capacity of a collection increases dynamically i.e. when we keep adding new elements, then the size of the collection keeps increasing automatically. Every collection class has three constructors and the behavior of collections will be as following when created using a different constructor.

1. **Default Constructor:** It Initializes a new instance of the collection class that is empty and has the default initial capacity as zero which becomes four after adding the first element and whenever needed the current capacity becomes double.
2. **Collection (int capacity):** This constructor initializes a new instance of the collection class that is empty and has the specified initial capacity, here also when the requirement comes current capacity doubles.
3. **Collection (Collection):** It Initializes a new instance of the collection class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied, here also when the requirement comes current capacity doubles.

**ArrayList in C#**

**ArrayList in C# with Examples**

In this article, I am going to discuss the **ArrayList in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the [**Introduction of Collection in C#**](https://dotnettutorials.net/lesson/collections-csharp/). As part of this article, we are going to discuss the following important concepts of the ArrayList class.

1. **What is ArrayList in C#?**
2. **Understanding the Methods and Properties of ArrayList class in C#.**
3. **Examples of ArrayList class.**
4. **What is the difference between an Array and ArrayList?**

**What is ArrayList in C#?**

The **ArrayList in C#** is a collection class that works like an array but provides the facilities such as dynamic resizing, adding, and deleting elements from the middle of a collection. It implements the System.Collections.IList interface using an array whose size is dynamically increased as required.

**Methods and Properties of ArrayList Collection class in C#:**

The following are the methods and properties provided by the ArrayList collection class in C#.

1. **Add(object value):**This method is used to add an object to the end of the collection.
2. **Remove(object obj):**This method is used to remove the first occurrence of a specific object from the collection.
3. **RemoveAt(int index):**This method takes the index position of the elements and removes that element from the collection.
4. **Insert(int index, Object value):**This method is used to inserts an element into the collection at the specified index.
5. **Capacity:** This property gives you the capacity of the collection means how many elements you can insert into the collection.

**Example of ArrayList Collection class in C#:**

Let us see an example by using the above methods and properties of the ArrayList class. The code is self-explained so please go through the comments.

**using** *System;*

**using** *System.Collections;*

**namespace** *ArrayListCollection*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Createing ArrayList collection using default constructor

ArrayList al = new ArrayList**()**;

Console.WriteLine**(**"Initial Capacity: " + al.Capacity**)**;

al.Add**(**10**)**;

Console.WriteLine**(**"Capacity after adding first item: " + al.Capacity**)**;

al.Add**(**"hello"**)**;

al.Add**(true)**;

al.Add**(**3.14f**)**;

Console.WriteLine**(**"Capacity after adding fourth item: " + al.Capacity**)**;

al.Add**(**'A'**)**;

Console.WriteLine**(**"Capacity after adding 5th element: " + al.Capacity**)**;

//Printing the ArrayList elements using for loop

**for** **(int** i = 0; i **<** al.Count; i++**)**

**{**

Console.Write**(**al**[**i**]** + " "**)**;

**}**

Console.WriteLine**()**;

//Removing the values from the middle of the array list

//here we are removing by value

al.Remove**(true)**;

//You can also remove element by using index position

// al.RemoveAt(2);

//Printing the ArrayList elements using foreach loop after

// removing an element from the collection

**foreach** **(object** obj in al**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//inserting values into the middle of the array list collection

al.Insert**(**2, **false)**;

// Printing the values of the collection using foreach loop after

// inserting an element into the middle of the collection

**foreach** **(object** obj in al**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

// creating new ArrayList collection by passing the old

// array list as parameter

ArrayList coll = new ArrayList**(**al**)**;

Console.WriteLine**(**"Initial capacity of new array list collection:" + coll.Capacity**)**;

// Printing the values of the new array list collection using foreach loop

**foreach** **(object** obj in coll**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

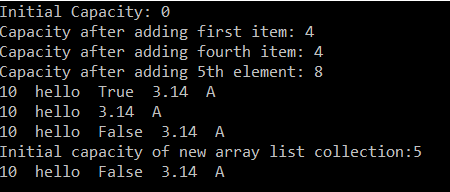
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**What is the difference between an array and an Array List in C#?**

This is one of the frequently asked interview questions in C#. So let us discuss the difference between an array and ArrayList.

**Array:**

1. Fixed Length
2. Cannot insert into the middle
3. Cannot delete from middle

**ArrayList:**

1. Variable Length
2. Can insert an element into the middle of the collection
3. Can delete element from the middle of the collection

**Hashtable in C#**

**Hashtable in C# with Examples**

In this article, I am going to discuss the **Hashtable in C#** with Examples. Please read our previous article where we discussed the [**ArrayList Collection in C#**](https://dotnettutorials.net/lesson/arraylist-collection-csharp/)with examples. Before understanding the Hashtable in C#, let us first understand the problems that we face with Array and ArrayList collection in C#.

**Problems with Array and ArrayList Collection in C#:**

In the case of [**Array**](https://dotnettutorials.net/lesson/arrays-csharp/)and [**ArrayList**](https://dotnettutorials.net/lesson/arraylist-collection-csharp/)in C#, we can access the elements from the collection using a key. That key is nothing but the index position of the elements which is starts from zero (0) to the number of elements – 1. But in reality, it’s very difficult to remember the index position of the element in order to access the values.

For example, let’s say we have an employee array that contains the name, address, mobile, dept no, email id, employeeid, salary, location, etc. Now if I want to know the email id or dept no of the employee then it’s very difficult for me. This is because we need to access the value by using the index position. The following example shows this. Here we create a collection using ArrayList and then we are accessing the Location and EmailId by using the index position.

**namespace** *HasntableExample*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ArrayList al = new ArrayList**()**;

al.Add**(**1001**)**; //EId

al.Add**(**"James"**)**; //Name

al.Add**(**"Manager"**)**; //Job

al.Add**(**3500**)**; //Salary

al.Add**(**"Mumbai"**)**; //Location

al.Add**(**"IT"**)**; //Dept

al.Add**(**"a@a.com"**)**; // Emailid

//I want to print the Location, index position is 4

Console.WriteLine**(**"Location : " + al**[**4**])**;

//I want to print the Email ID, index position is 6

Console.WriteLine**(**"Emaild ID : " + al**[**6**])**;

Console.ReadKey**()**;

**}**

**}**

**}**

If you have a huge number of elements in the collection, then it is very difficult to remember the index position of each element. So, it would not be nice, instead of using the index position of the element, if we can access the elements by using a key. This is where **Hashtable in C#** comes into the picture.

**What is a Hashtable in C#?**

The **Hashtable in C#** is a collection that stores the element in the form of “**Key-Value Pairs**”. The data in the Hashtable are organized based on the hash code of the key. The key in the HashTable is defined by us and more importantly, that key can be of any data type. Once we created the Hashtable collection, then we can access the elements by using the keys. The Hashtable class comes under the **System.Collections** namespace.

The Hashtable computes a hash code for each key. Then it uses that hash code to look up the elements very quickly which increases the performance of the application. Let us see an example for a better understanding of the Hashtable in C#.

**using** *System;*

**using** *System.Collections;*

**namespace** *HasntableExample*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating Hashtable collection with default constructor

Hashtable ht = new Hashtable**()**;

//Adding elements to the Hash table using key value pair

ht.Add**(**"EId", 1001**)**; //Here key is Eid and value is 1001

ht.Add**(**"Name", "James"**)**; //Here key is Name and value is James

ht.Add**(**"Job", "Developer"**)**;

ht.Add**(**"Salary", 3500**)**;

ht.Add**(**"Location", "Mumbai"**)**;

ht.Add**(**"Dept", "IT"**)**;

ht.Add**(**"EmailID", "a@a.com"**)**;

//Printing the keys and their values using foreach loop

Console.WriteLine**(**"Printing using Foreach loop"**)**;

**foreach** **(object** obj in ht.Keys**)**

**{**

Console.WriteLine**(**obj + " : " + ht**[**obj**])**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Printing using Keys"**)**;

//I want to print the Location by using the Location key

Console.WriteLine**(**"Location : " + ht**[**"Location"**])**;

//I want to print the Email ID by using the EmailID key

Console.WriteLine**(**"Emaild ID : " + ht**[**"EmailID"**])**;

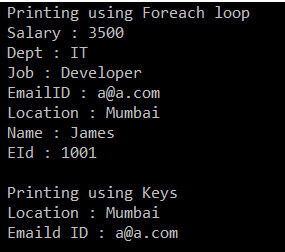
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



When you are working with the Hashtable, then you need to understand two important methods are as follows.

**ContainsKey() Method:**

The ContainsKey() method of the Hashtable is used to check if a given key is present in the Hashtable or not. The following is the syntax for using the ContainsKey() method. If the given key is present in the collection then it will return true else it will return false.   
**Hashtable.Containskey(key)**

**ContainsValue() Method:**

The ContainsValue() Method of the Hashtable class is used to check if a value is present in the Hashtable or not. The following is the syntax for using the ContainsValue() Method. If the given value is present in the collection then it will return true else it will return false.  
**Hashtable.ContainsValue(value)**

**Let us see an example for a better understanding of the above two methods:**

**using** *System;*

**using** *System.Collections;*

**namespace** *HasntableExample*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating Hashtable collection with default constructor

Hashtable ht = new Hashtable**()**;

//Adding elements to the Hash table using key value pair

ht.Add**(**"EId", 1001**)**; //Here key is Eid and value is 1001

ht.Add**(**"Name", "James"**)**; //Here key is Name and value is James

ht.Add**(**"Job", "Developer"**)**;

ht.Add**(**"Salary", 3500**)**;

ht.Add**(**"Location", "Mumbai"**)**;

ht.Add**(**"Dept", "IT"**)**;

ht.Add**(**"EmailID", "a@a.com"**)**;

//Checking the key using the ContainsKey methid

Console.WriteLine**(**"Is EmailID Key Exists : " + ht.ContainsKey**(**"EmailID"**))**;

Console.WriteLine**(**"Is Department Key Exists : " + ht.ContainsKey**(**"Department"**))**;

//Checking the value using the ContainsValue method

Console.WriteLine**(**"Is Mumbai value Exists : " + ht.ContainsValue**(**"Mumbai"**))**;

Console.WriteLine**(**"Is Technology value Exists : " + ht.ContainsValue**(**"Technology"**))**;

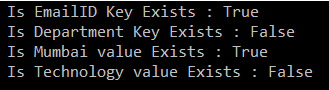
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Stack in C#**

**Stack in C# with Examples**

In this article, I am going to discuss the **Stack in C#** with examples. Please read our previous article before proceeding to this article where we discussed [**Hashtable in C#**](https://dotnettutorials.net/lesson/hashtable-csharp/) with examples. As part of this article, we are going to discuss the following important pointers related to Stack in C#.

1. **What is a Stack in C#?**
2. **How does a stack work?**
3. **Understanding the different methods of Stack in C#?**

**What is Stack in C#?**

The **Stack in C#** is a non-generic collection class that works in the **LIFO (Last In First Out)** principle. So, we need to use the Stack Collection in C#, when we need last in first out access to the items of a collection. That means the item which is added last will be removed first. When we add an item into the stack, then it is called pushing an item. Similarly when we remove an item from the stack then it is called popping an item. The Stack class belongs to the **System.Collections** namespace.

Let us understand the LIFO principle with an example. Imagine we have a stack of plates with each plate added on top of each other. The last plate which is added to the stack will be the first one to remove from the stack. It is not possible to remove a plate from the middle of the stack.

**Note:** The **Stack Collection in C#** allows both null and duplicate values.

**Methods of Stack class in C#:**

**Push():** The push() method is used to Inserts an object on top of the Stack.  
**Syntax: void Stack.Push(Object obj)**

**Pop():** The pop() method is used to remove and return the object at the top of the Stack. If there is no object (or element) present in the stack and if you are trying to remove an item or object from the stack using the pop() method then it will throw an exception i.e. **System.InvalidOperationException**  
**Syntax: Object stack.pop()**

**Peek():** The peek() method is used to return the object from the top of the Stack without removing it. If there is no object (or element) present in the stack and if you are trying to return an item (object) from the stack using the peek() method then it will throw an exception i.e. **System.InvalidOperationException**  
**Syntax: object Stack.Peek()**

**Example: Stack in C#**

Let us understand the methods of the Stack class using an example in C#.

**using** *System;*

**using** *System.Collections;*

**namespace** *StackCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating a stack collection

Stack s = new Stack**()**;

//Adding item to the stack using the push method

s.Push**(**10**)**;

s.Push**(**"hello"**)**;

s.Push**(**3.14f**)**;

s.Push**(true)**;

s.Push**(**67.8**)**;

s.Push**(**'A'**)**;

//Printing the stack items using foreach loop

**foreach** **(object** obj in s**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Removing and returning an item from the stack

//using the pop method

Console.WriteLine**(**s.Pop**())**;

Console.WriteLine**()**;

//Printing item after removing the last added item

**foreach** **(object** obj in s**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Returning the last item from the stack without removing it

//by using the peek method

Console.WriteLine**(**s.Peek**())**;

Console.WriteLine**()**;

//Printing the items after using the Peek method

**foreach** **(object** obj in s**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

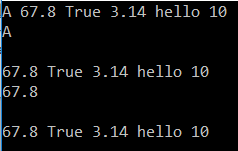
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Let us discuss some other methods and properties of Stack Class:**

**Count:** The Count property of the Stack class is used to return the number of elements present in the Stack.  
**Syntax:** **Stack.Count**

**Contains():** The Contains() method of the Stack class is used to check whether an element is present in the Stack or not. If it presents, then it will return true else it will return false.  
**Syntax:** **Stack.Contains(element)**

**Clear():** The Clear() method of the Stack class is used to remove all the elements from the stack.  
**Syntax:** **Stack.Clear()**

**Example: Let us see an example for a better understanding of the above methods and properties**

**using** *System;*

**using** *System.Collections;*

**namespace** *StackCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating a stack collection

Stack s = new Stack**()**;

//Adding item to the stack using the push method

s.Push**(**20**)**;

s.Push**(**"hi"**)**;

s.Push**(**3.14f**)**;

s.Push**(true)**;

s.Push**(**12.3**)**;

s.Push**(**'P'**)**;

//Printing the stack items using foreach loop

**foreach** **(object** obj in s**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Using Count property to get the number of items

//present in the collection

Console.WriteLine**(**$"No of Elements in the Collection : {s.Count}"**)**;

Console.WriteLine**()**;

//Using the Contains method to check whether an item is present or not

Console.WriteLine**(**$"Is the value hi present in the collection : {s.Contains("hi")}"**)**;

Console.WriteLine**()**;

//Removing all the items from the collection using Clear() method

s.Clear**()**;

Console.WriteLine**(**$"No of Elements in the Collection after Clear() method : {s.Count}"**)**;

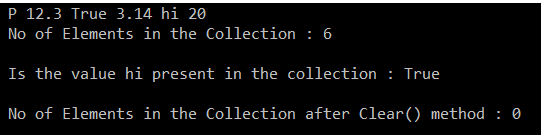
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Queue in C#**

**Queue in C# with Examples**

In this article, I am going to discuss the **Queue in C#** with examples. Please read our previous article before proceeding to this article where we discussed the non-generic collection [**Stack in C#**](https://dotnettutorials.net/lesson/stack-collection-csharp/)with some examples. As part of this article, we are going to discuss the following pointers of the Queue collection class in C#.

1. **What is a Queue in C#?**
2. **How does the Queue work?**
3. **Understanding the different methods of Queue Collection class in C#.**

**What is a Queue in C#?**

The Queue in C# is a non-generic collection class that works in the **FIFO (First In First Out)** principle. So, we need to use the Queue Collection in C#, when we need the first in first out access to the items of a collection. That means the item which is added first will be removed first from the collection. When we add an item into the queue collection, it is called enqueuing an item. Similarly when we remove an item from the queue collection then it is called dequeuing an item. The Queue class belongs to the **System.Collections** namespace.

Let us understand the FIFO principle with an example. Imagine a queue of people waiting for the ticket in a cinema hall. Normally, the first person who enters the queue will be the first person to get the ticket from the counter. Similarly, the last person who enters the queue will be the last person to get the ticket from the counter.

**Note:** The non-generic **Queue Collection class in C#** allows both null and duplicate values.

**Methods of Queue class in C#:**

**Enqueue():** This method is used to add an item (or object) to the end of the Queue.  
**Syntax:** **void Queue.Enqueue(object obj)**

**Dequeue():** The Dequeue() method of the Queue class is used to Remove and return the object from the beginning of the Queue. If there is no object (or element) present in the Queue and if we are trying to remove an item or object from the Queue using the pop() method then it will throw an exception i.e. **System.InvalidOperationException**  
**Syntax:** **object Queue.Dequeue()**

**Peek():**The peek() method of the Queue class is used to return the oldest object i.e. the object present at the start of the Queue without removing it. If there is no object (or element) present in the Queue and if we are trying to return an item (object) from the Queue using the peek() method then it will throw an exception i.e. **System.InvalidOperationException**  
**Syntax: object Queue.Peek()**

**Example: Let us understand the above methods of Queue class with an example.**

**using** *System;*

**using** *System.Collections;*

**namespace** *QueueCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating a queue collection

Queue q = new Queue**()**;

//Adding item to the queue using the Enqueue method

q.Enqueue**(**10**)**;

q.Enqueue**(**"hello"**)**;

q.Enqueue**(**3.14f**)**;

q.Enqueue**(true)**;

q.Enqueue**(**67.8**)**;

q.Enqueue**(**'A'**)**;

//Printing the queue items using foreach loop

**foreach** **(object** obj in q**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Removing and returning an item from the queue

//using the Dequeue method

Console.WriteLine**(**q.Dequeue**())**;

Console.WriteLine**()**;

//Printing item after removing the first added item

**foreach** **(object** obj in q**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Returning the first item from the queue without removing it

//by using the peek method

Console.WriteLine**(**q.Peek**())**;

Console.WriteLine**()**;

//Printing the items after using the Peek method

**foreach** **(object** obj in q**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

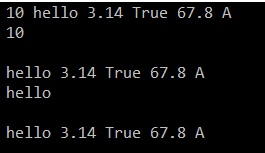
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Let us discuss some other important methods and properties of Queue Class:**

**Count:** The Count property of the Queue class is used to return the number of elements present in the Queue Collection.  
**Syntax:** **Queue.Count**

**Contains():** The Contains() method of the Queue class is used to check whether an object (element) is present in the Queue or not. If it presents, then it will return true else it will return false.  
**Syntax:** **Queue.Contains(element)**

**Clear():** The Clear() method of the Queue class is used to remove all the elements from the queue collection.  
**Syntax: Queue.Clear()**

**Example: Let us see an example for a better understanding of the above methods and properties of Queue Class**

**using** *System;*

**using** *System.Collections;*

**namespace** *QueueCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating a queue collection

Queue q = new Queue**()**;

//Adding item to the qieue using the Enqueue method

q.Enqueue**(**10**)**;

q.Enqueue**(**"hi"**)**;

q.Enqueue**(**3.14f**)**;

q.Enqueue**(true)**;

q.Enqueue**(**67.8**)**;

q.Enqueue**(**'A'**)**;

//Printing the queue items using foreach loop

**foreach** **(object** obj in q**)**

**{**

Console.Write**(**obj + " "**)**;

**}**

Console.WriteLine**()**;

//Using Count property to get the number of items

//present in the queue collection

Console.WriteLine**(**$"No of Elements Present in the Collection : {q.Count}"**)**;

Console.WriteLine**()**;

//Using the Contains method to check whether an item is present or not

Console.WriteLine**(**$"Is the value hi present in the collection : {q.Contains("hi")}"**)**;

Console.WriteLine**()**;

//Removing all the items from the collection using Clear() method

q.Clear**()**;

Console.WriteLine**(**$"No of Elements in the Collection after Clear() method : {q.Count}"**)**;

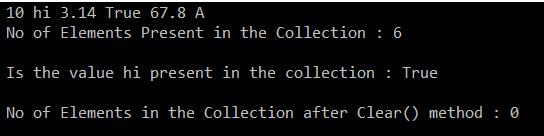
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Advantages and Disadvantages of Non-Generic Collection in C#**

**Advantages and Disadvantages of Non-Generic Collection in C#**

In this article, I am going to discuss the **Advantages and Disadvantages of Non-Generic Collection in C#**. Please read our previous article where we discussed [**Queue in C#**](https://dotnettutorials.net/lesson/queue-collection-class-csharp/) with Examples. Here we will discuss the advantages and disadvantages of using the ArrayList collection class which can also be applied to other non-generic collection classes such as Stack, Queue, and Hashtable, etc.

**Advantages of using ArrayList Collection class in C#:**

As we already discussed the non-generic collection classes can grow in size automatically when we add items into the collection and this is the advantage. Let us prove this with an example.

In the following example, we create a collection i.e. Numbers of the type ArrayList with the initial size 3. But we add the 4 elements into the collection, we did not get any errors. It works as expected. Hence, it proves the collection like **ArrayList, Stack, Queue, Hashtable**, etc. can grow in size dynamically when we add items into the collection. If this is an integer array, we will get the index out of the bound run time exception when we add the 4th element into the collection.

**Example:**

**namespace** *CollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ArrayList Numbers = new ArrayList**(**3**)**;

Numbers.Add**(**100**)**;

Numbers.Add**(**200**)**;

Numbers.Add**(**300**)**;

Numbers.Add**(**400**)**;

**foreach(int** Number in Numbers**)**

**{**

Console.Write**(**Number + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Advantages and Disadvantages of Collection

The non-generic collection classes such as ArrayList, Stack, Queue, and Hashtable provide several useful methods to add and remove items to the collection as well as they also provide some methods using which we can add or remove items from the middle of a collection and this is the other benefits which we get in non-generic collection classes in C#.

**Disadvantages of using ArrayList in C#:**

The non-generic collection classes such as **ArrayList, Stack, Queue, Hashtable,** etc operate on the object data type. As they operator on object data type hence they are loosely typed. Loosely typed means you can store any type of value into the collection. Because of this loosely typed nature, we may get runtime errors.

Not only we get run time errors because of the loosely-typed nature, but it also affects the performance of the application due to boxing and unboxing. Let us understand this with an example.

In the following example, we create a non-generic collection i.e. Numbers of the type ArrayList with the initial size 2. Then we are storing two elements such as 100 and 200. These two items 100 and 200 are integers as well as value types.

The collection classes belong to **System.Collections** namespace operates on the object data type. The object data type in C# is a reference data type. So the value that we storing in the collection is converted to reference type. So in our example, the values 100 and 200 are boxed and converted into the reference type. In our example, we just stored two values. Consider a scenario where we need to store 1000 integers values. Then all the 1000 integers are boxed, meaning they are converted into reference types and then stored in the collection.

Similarly, when we want to retrieve the items from the collection, then again we need to covert the object type back to the integer type meaning performing an unboxing. So this unnecessary boxing and unboxing happen behind the scenes every time we add and retrieve value types to the collection. So if you are operating on a large collection of value types then it may degrade the performance of your application.

**Example:**

**namespace** *CollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ArrayList Numbers = new ArrayList**(**2**)**;

// Boxing happens - Converting Value type (100, 200) to reference type

// Means Integer to object type

Numbers.Add**(**100**)**;

Numbers.Add**(**200**)**;

// Unboxing happens - 100 and 200 stored as object type

// now conveted to Integer type

**foreach** **(int** Number in Numbers**)**

**{**

Console.Write**(**Number + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**So in short: Problems with non-generic Collection Classes in C#:**

Non-generic collection classes are not type-safe as they operate on object data type so they can store any type of value.

1. Array is type-safe
2. Array List, HashTable, Stack, and Queue are not type-safe

For example, if I want to store n no of integer values

1. I cannot go with an array as arrays are fixed lengths. In this case, the length is unknown
2. I can go with array list or hash table but if we go with array list or hash table then there is a chance of storing other types of values as they are not type-safe as they operate on the object data type

So the solution is Generic collections in C#.

1. **Array**: Type-safe but fixed length
2. **Collections**: Auto Resizing but not type-safe
3. **Generic Collections**: Typesafe and auto-resizing

**Generic Collections in C#**

**Generic Collections in C#**

In this article, I am going to give a brief introduction to **Generic Collections in C#**. Please read our previous article where we discussed the [**Advantages and Disadvantages of the Non-Generic Collection**](https://dotnettutorials.net/lesson/advantages-and-disadvantages-of-collection/)in C#. As part of this article, first, we will discuss the problems of non-generic collections and then we will discuss how to overcome the non-generic collection problems with generic collections in C#. Finally, we will discuss some of the generic collection classes examples in C#.

The **Generic Collections in C#** are introduced as part of C# 2.0. You can consider this Generic collection as an extension to the non-generic collection classes which we have already discussed in our previous articles such as [**ArrayList**](https://dotnettutorials.net/lesson/arraylist-collection-csharp/), [**Hashtable**](https://dotnettutorials.net/lesson/hashtable-csharp/), [**Stack**](https://dotnettutorials.net/lesson/stack-collection-csharp/), and [**Queue**](https://dotnettutorials.net/lesson/queue-collection-class-csharp/).

**Problems with Non-Generic Collections in C#**

The non-generic collection classes such as ArrayList, Hashtable, Stack, and Queue are worked on the object data type. That means the elements are added to the collection are of an object type. As these non-generic collection classes worked on object data type, we can store any type of values that may lead to a runtime exception due to type mismatch. But with **Generic Collections in C#**, now are able to store a specific type of data (whether a primitive type or a reference type) which provides type safety by eliminating the type mismatch at run time.

The second problem with non-generic collection classes is that we get performance overhead. The reason for this is boxing and unboxing. As we already discussed these collection classes are worked on the object data type. So if we are storing value type data into the collection, then those value type data are first converted into object type and then only store into the collection which is nothing but performing boxing. Similarly, if we want to retrieve the data from the collection, then we need to convert the data from object type to value type means performing unboxing. Due to this boxing and unboxing, we get poor performance if our collection is a big one.

**Note:** Boxing means converting a value type to an object type and Unboxing means converting object type back to the value type.

**The solution to the Non-Generic Collection Problem**

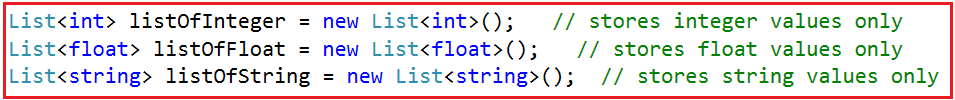
The above two problems of non-generic collections are overcome by the Generic collections in C#. The dot net framework has re-implemented all the existing collection classes such as **ArrayList, Hashtable, Stack,**and **Queue**, etc. in Generic Collections such as **ArrayList<T>, Dictionary<TKey, TValue>, Stack<T>**and **Queue<T>**. Here T is nothing but the type of values that we want to store in the collection. So. the most important point that you need to remember is while creating the objects of Generic Collection classes, you need to explicitly provide the type of values that you are going to store into the collection.

The Generic Collection classes are implemented under the **System.Collections.Generic** namespace. The classes which are present in this namespace are as follows.

1. **Stack<T>,**
2. **Queue<T>,**
3. **LinkedList<T>,**
4. **SortedList<T>,**
5. **List<T>,**
6. **Dictionary<TKey, Tvalue>**

**Note:**Here the <T> refers to the type of values we want to store under them.

**Examples:**



It is also possible to store a user-defined type like a class type or structure type as shown below  
**List<Customer> listCustomer = new List<Customer> ();**  
Assume the **Customer** is a user-defined class type that represents an entity Customer, Here we can store the customer objects within the listCustomer collection where each customer object can internally represent different attributes of customer like id, name, balance, city, state, etc.

**Generic Collections in C#:**

The **Generic Collections in C#** are strongly typed. The strongly typed nature allows these collection classes to store only one type of value into it. This not only eliminates the type mismatch at runtime but also we will get better performance as they don’t require boxing and unboxing while they store value type data.

So, It is always a preferable and a good programming choice to use the Generics Collection Classes in C# rather than using the Non-Generic Collection Classes.

**Generics in C#**

**Generics in C# with Examples**

In this article, I am going to discuss how to implement **Generics in C#** with examples. Please read our previous article where we discussed the [**Generic Collection**](https://dotnettutorials.net/lesson/generic-collections-csharp/) in C#. As part of this article, we are going to discuss the following pointers.

1. **Why do we need generics in C#?**
2. **What are Generics in C#?**
3. **Advantages of Generics in C#.**
4. **How to implement Generics in C#?**
5. **How to use Generics with class and its members?**

**Why do we need Generics in C#?**

Let us understand the need for Generics in C# with one example. Let us create a simple program to check whether two integer numbers are equal or not. The following code implementation is very straightforward. Here we created two classes with the name **ClsCalculator** and **ClsMain**. Within the **ClsCalculator**class, we have **AreEqual()** method which takes two integer values as the input parameter and then it checks whether the two input values are equal or not. If both are equal then it returns true else it will return false. And from the **ClsMain**class, we are calling the static **AreEqual()** method and showing the output based on the return value.

**namespace** *GenericsDemo*

**{**

**public** **class** ClsMain

**{**

**private** **static** **void** Main**()**

**{**

**bool** IsEqual = ClsCalculator.AreEqual**(**10, 20**)**;

**if** **(**IsEqual**)**

**{**

Console.WriteLine**(**"Both are Equal"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Both are Not Equal"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** ClsCalculator

**{**

**public** **static** **bool** AreEqual**(int** value1, **int** value2**)**

**{**

**return** value1 == value2;

**}**

**}**

**}**

The above **AreEqual()** method works as expected as it is and more importantly it will only work with the integer values as this is our initial requirement. Suppose our requirement changes, now we also need to check whether two string values are equal or not.

In the above example, if we try to pass values other than the integer values, then we will get a compile-time error. This is because the **AreEqual()** method of the **ClsCalculator**class is tightly bounded with the integer data type and hence it is not possible to invoke the AreEqual method other than the integer data type values. So, when we try to invoke the **AreEqual()** method by passing string values as shown below we get a compile-time error.

**bool Equal = ClsCalculator.AreEqual(“ABC”, “XYZ”);**

One of the ways to make the above AreEqual() method to accepts string type values as well as integer type values, we need to make use of the object datatype as the parameters. If we make the parameters of the AreEqual() method as Object type, then it is going to works with any data type.

**Note:** The most important point that you need to keep in remember is every .NET data type whether it is a primitive type of reference type, is directly or indirectly inherits from the **System.Object**data type.

**Modifying the Method to accepts any data type values:**

Let’s modify the AreEqual() method of the **ClsCalculator**class to use the Object data type as shown below.

**namespace** *GenericsDemo*

**{**

**public** **class** ClsMain

**{**

**private** **static** **void** Main**()**

**{**

// bool IsEqual = ClsCalculator.AreEqual(10, 20);

**bool** IsEqual = ClsCalculator.AreEqual**(**"ABC", "ABC"**)**;

**if** **(**IsEqual**)**

**{**

Console.WriteLine**(**"Both are Equal"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Both are Not Equal"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** ClsCalculator

**{**

//Now this method can accept any data type

**public** **static** **bool** AreEqual**(object** value1, **object** value2**)**

**{**

**return** value1 == value2;

**}**

**}**

**}**

That’s it. Run the application and you will see it is working as expected. Let’s see the problem of the above code implementation.

1. We get poor Performance due to boxing and unboxing. The object type needs to be converted to the value type.
2. Now, the AreEuqal() method is not type-safe. Now it is possible to pass a string value for the first parameter and an integer value for the second parameter.

Method Overloading to Achieve the same:

Another option is we need to overload the AreEqual method which will accept different types of parameters as shown below. As you can see in the below code, now we have created three methods with the same name but with different types of parameters. This is nothing but method overloading. Now, run the application and you will see everything is working as expected.

**namespace** *GenericsDemo*

**{**

**public** **class** ClsMain

**{**

**private** **static** **void** Main**()**

**{**

// bool IsEqual = ClsCalculator.AreEqual(10, 20);

// bool IsEqual = ClsCalculator.AreEqual("ABC", "ABC");

**bool** IsEqual = ClsCalculator.AreEqual**(**10.5, 20.5**)**;

**if** **(**IsEqual**)**

**{**

Console.WriteLine**(**"Both are Equal"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Both are Not Equal"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** ClsCalculator

**{**

**public** **static** **bool** AreEqual**(int** value1, **int** value2**)**

**{**

**return** value1 == value2;

**}**

**public** **static** **bool** AreEqual**(**string value1, string value2**)**

**{**

**return** value1 == value2;

**}**

**public** **static** **bool** AreEqual**(double** value1, **double** value2**)**

**{**

**return** value1 == value2;

**}**

**}**

**}**

The problem with the above code implementation is that we are repeating the same logic in each and every method. However, if tomorrow we need to compare two float or two long values then again we need to create two more methods.

**How to solve the above Problems?**

We can solve all the above problems with **Generics in C#**. With generics, we will make the AreEqual() method to works with different types of data types. Let us first modify the code implementation to use the generics and then we will discuss how it works.

**namespace** *GenericsDemo*

**{**

**public** **class** ClsMain

**{**

**private** **static** **void** Main**()**

**{**

//bool IsEqual = ClsCalculator.AreEqual<int>(10, 20);

//bool IsEqual = ClsCalculator.AreEqual<string>("ABC", "ABC");

**bool** IsEqual = ClsCalculator.AreEqual**<double>(**10.5, 20.5**)**;

**if** **(**IsEqual**)**

**{**

Console.WriteLine**(**"Both are Equal"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Both are Not Equal"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** ClsCalculator

**{**

**public** **static** **bool** AreEqual**<**T**>(**T value1, T value2**)**

**{**

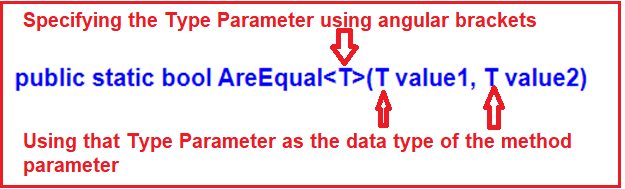
**return** value1.Equals**(**value2**)**;

**}**

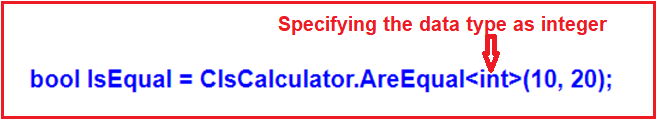
**}**

**}**

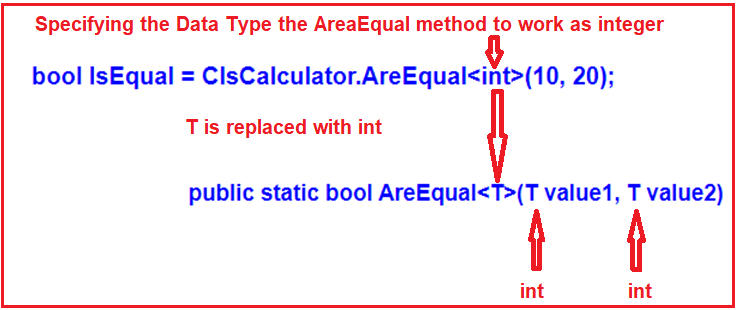
Here in the above example, in order to make the **AreEqual()**method generic (generic means the same method will work with the different data type), we specified the type parameter T using the angular brackets <T>. Then we use that type as the data type for the method parameters as shown in the below image.



At this point, if you want to invoke the above **AreEqual()** method, then you need to specify the data type on which the method should operate. For example, if you want to work with integer values, then you need to invoke the **AreEqual()** method by specifying int as the data type as shown in the below image using angular brackets.



The above AreEqual() generic method is working as follows:



If you want to work with the string values, then you need to call the AreEqual() method as shown below.  
**bool IsEqual= ClsCalculator.AreEqual<string>(“ABC”, “ABC”);**  
Now, I hope you understand the need and importance of Generics in C#.

**What is Generics in C#?**

As we already discussed in our [**previous article**](https://dotnettutorials.net/lesson/generic-collections-csharp/), the Generics in C# are introduced as part of C# 2.0. The Generics in C# allows us to define classes and methods which are decoupled from the data type. In other words, we can say that the Generics allow us to create classes using angular brackets for the data type of its members. At compilation time, these angular brackets are going to be replaced with some specific data types. In C#, the Generics can be applied to the following:

1. Interface
2. Abstract class
3. Class
4. Method
5. Static method
6. Property
7. Event
8. Delegates
9. Operator

**Advantages of Generics in C#**

1. It Increases the reusability of the code.
2. The Generics are type-safe. We will get the compile-time error if we try to use a different type of data rather than the one we specified in the definition.
3. We get better performance with Generics as it removes the possibilities of boxing and unboxing.

**How to use Generics with class and its members?**

Let us create a generic class with a generic constructor, generic member variable, generic property, and a generic method as shown below.

**using** *System;*

**namespace** *GenericsDemo*

**{**

//MyGenericClass is a Generic Class

**class** MyGenericClass**<**T**>**

**{**

//Generic variable

//The data type is generic

**private** T genericMemberVariable;

//Generic Constructor

//Constructor accepts one parameter of Generic type

**public** MyGenericClass**(**T **value)**

**{**

genericMemberVariable = **value**;

**}**

//Generic Method

//Method accepts one Generic type Parameter

//Method return type also Generic

**public** T genericMethod**(**T genericParameter**)**

**{**

Console.WriteLine**(**"Parameter type: {0}, value: {1}", typeof**(**T**)**.ToString**()**, genericParameter**)**;

Console.WriteLine**(**"Return type: {0}, value: {1}", typeof**(**T**)**.ToString**()**, genericMemberVariable**)**;

**return** genericMemberVariable;

**}**

//Generic Property

//The data type is generic

**public** T genericProperty **{** **get**; **set**; **}**

**}**

**}**

In the above example, we created the class **MyGenericClass** with **<T>**. The angular brackets (“<>”) indicate that the MyGenericClass class is a generic class and the type for this class is going to be defined later.

While creating the instance of this MyGenericClass class, we need to specify the type and the compiler will assign that type to T. In the following example, we use **int** as the data type:

**class** Program

**{**

**static** **void** Main**()**

**{**

MyGenericClass**<int>** integerGenericClass = new MyGenericClass**<int>(**10**)**;

**int** val = integerGenericClass.genericMethod**(**200**)**;

Console.ReadKey**()**;

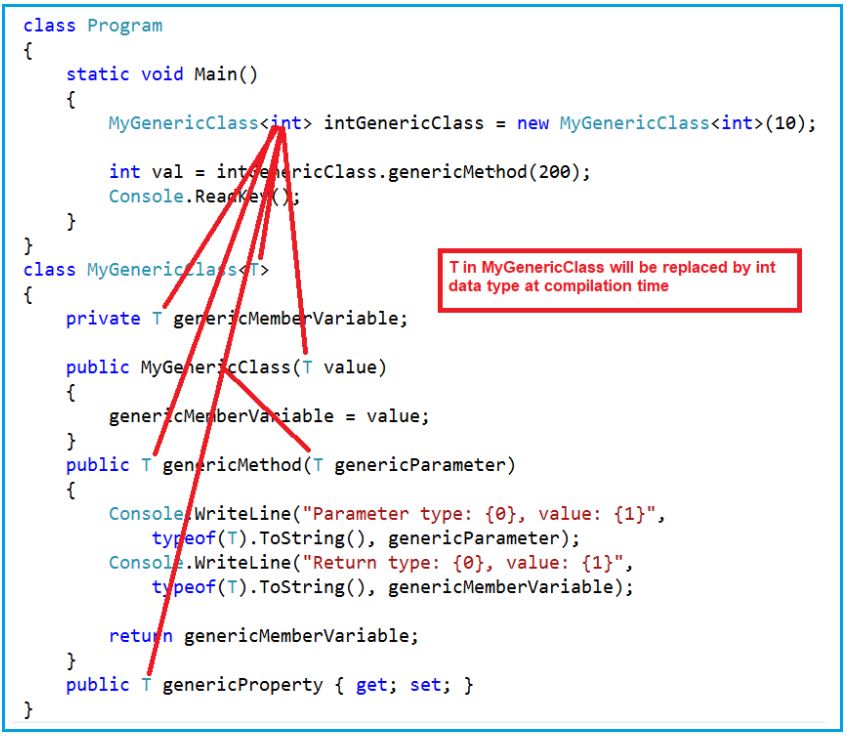
**}**

**}**

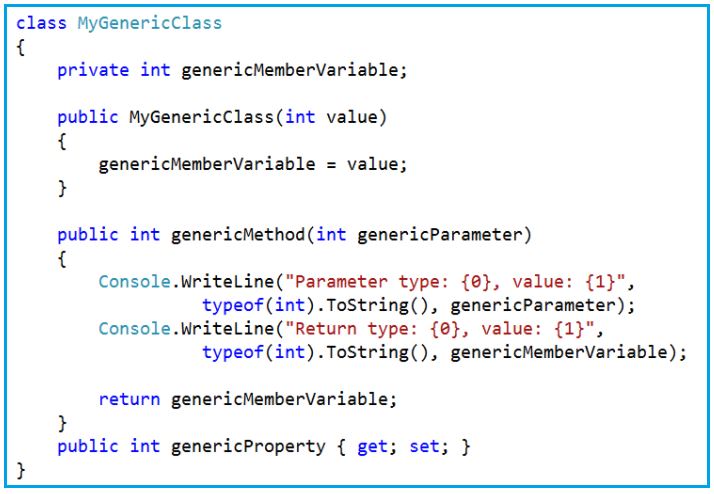
Run the application and it will give you the following output.

Generics in C#

**The following diagram shows how the T will be replaced with the int data type by the compiler.**



**The compiler will compile the above class as shown in the below image**



At the time of instantiation, we can use any type as per our requirement. If we want to use a string type, then we need to instantiate the class as shown below

**class** Program

**{**

**static** **void** Main**()**

**{**

MyGenericClass**<**string**>** stringGenericClass = new MyGenericClass**<**string**>(**"Hello Generic World"**)**;

stringGenericClass.genericProperty = "This is a generic property example.";

string result = stringGenericClass.genericMethod**(**"Generic Parameter"**)**;

Console.ReadKey**()**;

**}**

**}**

**Output:**

Generics in C#

I hope you understand the concept of Generics in C#. The generics are extremely used by the collection classes which belong to **System.Collections.Generic** namespace.

**Generic List in C#**

**Generic List in C# with Examples**

In this article, I am going to discuss the **Generic List in C#** with examples. Please read our previous article where we discussed the [**Generics in C#**](https://dotnettutorials.net/lesson/generics-csharp/) with examples. At the end of this article, you will understand what exactly List in C# and when and how to use Generic List with examples.

**What is Generic List in C#?**

The **Generic List in C#** is a collection class that is present in **System.Collections.Generic** namespace. The List Collection class is one of the most widely used generic collection classes in real-time applications. This Generic List collection class represents a strongly typed list of objects which can be accessed by using the index. It also provides methods that can be used for search, sort and manipulate the list items.

We can create a collection of any type by using the generic list class in C#. For example, if we want then we can create a list of strings, a list of integers, and even though it is also possible to create a list of the user-defined complex type such as a list of customers, a list of products, etc. The most important point that we need to keep in mind is the size of the collection grows automatically when we add items into the collection.

**Methods and Properties of Generic List Collection class in C#:**

The following are some of the useful methods and properties of the List collection class in C#.

1. **Add(T value):** This method is used to add an item to the end of the list collection.
2. **Remove(T value):** This method is used to remove the first occurrence of a specific item from the collection.
3. **RemoveAt(int index):** This method takes the index position of the elements and then removes that element from the collection.
4. **Insert(int index, T value):** This method is used to inserts an element into the collection at a specified index position.
5. **Capacity:** This property is used to return the capacity of the collection means how many elements you can insert into the collection.

**Example of Generic List in C#:**

Let us see an example to understand the above methods and properties of Generic List in C#. So, basically here, we are creating a list of integer types and then adding items to this collection using the Add method and then performing different types of operations. The following example is self-explained. So please go through the comments.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *GenericListCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Createing an interger list

//it is going to store only integer value

List**<int>** integerList = new List**<int>()**;

Console.WriteLine**(**"Initial Capacity: " + integerList.Capacity**)**;

integerList.Add**(**10**)**;

Console.WriteLine**(**"Capacity after adding first item: " + integerList.Capacity**)**;

integerList.Add**(**20**)**;

integerList.Add**(**30**)**;

integerList.Add**(**40**)**;

Console.WriteLine**(**"Capacity after adding fourth item: " + integerList.Capacity**)**;

integerList.Add**(**60**)**;

Console.WriteLine**(**"Capacity after adding 5th element: " + integerList.Capacity**)**;

//Printing the List items using for loop

Console.WriteLine**(**"Printing the List items using for loop:"**)**;

**for** **(int** i = 0; i **<** integerList.Count; i++**)**

**{**

Console.Write**(**integerList**[**i**]** + " "**)**;

**}**

Console.WriteLine**()**;

//Removing the values from the middle of the list

//here we are removing by value

integerList.Remove**(**30**)**;

//You can also remove element by using index position

//integerList.RemoveAt(2);

//Printing the list items using foreach loop after

//removing an element from the collection

Console.WriteLine**(**"List items after removing the value 30 :"**)**;

**foreach** **(int** item in integerList**)**

**{**

Console.Write**(**item + " "**)**;

**}**

Console.WriteLine**()**;

//inserting values into the middle of the list collection

integerList.Insert**(**2, 25**)**;

// Printing the values of the collection using foreach loop after

// inserting an element into the middle of the collection

Console.WriteLine**(**"List items after inserting the value 25 in the index 2"**)**;

**foreach** **(int** item in integerList**)**

**{**

Console.Write**(**item + " "**)**;

**}**

Console.WriteLine**()**;

// creating new ArrayList collection by passing the old

// array list as parameter

List**<int>** newIntegerList = new List**<int>(**integerList**)**;

Console.WriteLine**(**"Initial capacity of new list collection:" + newIntegerList.Capacity**)**;

// Printing the values of the new list collection using foreach loop

Console.WriteLine**(**"Printing the new List items which is created from the old list"**)**;

**foreach** **(int** item in newIntegerList**)**

**{**

Console.Write**(**item + " "**)**;

**}**

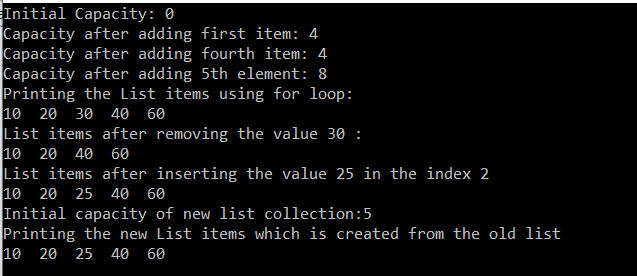
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**List Collection with Complex Type in C#:**

Let us see an example of the List Collection class with a complex type. As you can see in the below code, we have a class called Employee. Then we create few employee objects and then we create a collection of type Employee and storing all the employee objects into the collection. Finally, we are performing different types of operations using the methods provided by the List<T> generic collection class. The following example code is self-explained. So please go through the comments.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *ListCollectionDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

// Create Employee Objects

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 5500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

// Create a List of Employees

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

// We can retrieve the Items from a list collection by using index.

// The following line of code will retrieve the employee from the list.

// The List index is also 0 based.

Employee emp = listEmployees**[**0**]**;

Console.WriteLine**(**"Retrive the First employee by index"**)**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

Console.WriteLine**()**;

// retrieving the list using for loop

Console.WriteLine**(**"Retriving the list using for loop"**)**;

**for** **(int** i = 0; i **<** listEmployees.Count; i++**)**

**{**

Employee employee = listEmployees**[**i**]**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

// retrieving the list using foreach loop

Console.WriteLine**(**"Retriving the list using foreach loop"**)**;

**foreach** **(**Employee e in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

e.ID, e.Name, e.Gender, e.Salary**)**;

**}**

Console.WriteLine**()**;

// inserting an employee into the index position 1.

listEmployees.Insert**(**1, emp4**)**;

// retrieving the list after inserting the employee in index position 1

Console.WriteLine**(**"Retriving the list after inserting new employee in index 1"**)**;

**foreach** **(**Employee e in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

e.ID, e.Name, e.Gender, e.Salary**)**;

**}**

Console.WriteLine**()**;

//If you want to get the index postion of a specific employee

//then use Indexof() method as shown below

Console.WriteLine**(**"Index of emp3 object in the List = " +

listEmployees.IndexOf**(**emp3**))**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

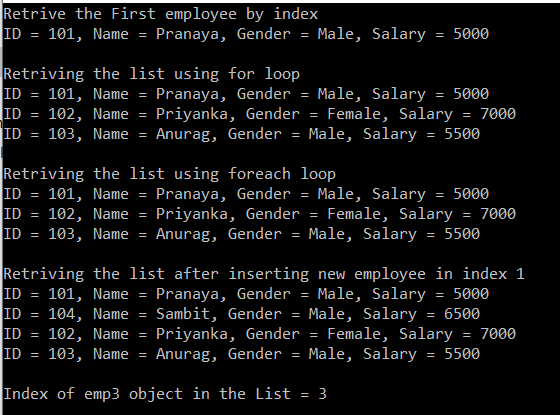
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**Note:** All the generic classes in C# are strongly typed. That means if we have created a List of type Employee, then we can only add objects of type Employee into the list. If we try to add an object of a different type, then we will get a compile-time error.

**Important methods of generic list collection class in C#:**

The Generic List in C# provides a lot of useful methods that we can use to perform different operations on a collection of List Type. The List Collection class provides the following functions.

**Contains() Method:**  
The Contains() method is used to determine whether the specified item exists or not in the list. If the specified item exists then it will return true else return false.

**Exists() Method:**  
The Exists() method is also used to check or determine whether an item exists or not in a list based on a condition. If the item exists then it will return true else it will return false.

**Find() Method:**  
The Find() method is used to find the first element from a list based on a condition that is specified by a lambda expression.

**FindLast() Method:**  
The FindLast() method is used to searches for an element that matches the conditions specified by a predicate. If it found any elements with that specified condition then it returns the Last matching element from the list.

**FindAll() Method:**  
The FindAll() method is used to retrieves all the elements from a list that matches the conditions specified by a predicate.

**FindIndex() Method:**

The FindIndex() method is used to return the index position of the first element that matches the conditions specified by a predicate. The point that you need to remember is the index here in generic collections is zero-based. This method returns -1 if an element that matches the specified conditions is not found. There are 2 other overloaded versions of this method is available, one of the overload versions allows us to specify the range of elements to search within the list.

**FindLastIndex() Method:**

The FindLastIndex() Method searches for an element in the list that matches the condition specified by the lambda expression and then returns the index of the last occurrence of the item within the list. There are 2 other overloaded versions of this method is available, one of the overload versions allows us to specify the range of elements to search within the list.

**Let us understand all the above methods of List Collection class in C# with an example.**

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *ListCollectionDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

// Create Employee Objects

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 5500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

//Creating a list of type Employee

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

// use Contains method to check if an item exists or not in the list

**if** **(**listEmployees.Contains**(**emp2**))**

**{**

Console.WriteLine**(**"Employee2 object exists in the list"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Employee2 object does not exist in the list"**)**;

**}**

Console.WriteLine**(**""**)**;

// Use Exists method when you want to check if an item exists or not

// in the list based on a condition

**if** **(**listEmployees.Exists**(**x =**>** x.Name.StartsWith**(**"P"**)))**

**{**

Console.WriteLine**(**"List contains Employees whose name starts with P"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"List does not contain any employee whose name starts with P"**)**;

**}**

Console.WriteLine**(**""**)**;

// Use Find() method, if you want to searche an element by a conditions

Employee emp = listEmployees.Find**(**employee =**>** employee.Gender == "Male"**)**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

Console.WriteLine**(**""**)**;

// Use FindLast() method when you want to searche an item by a conditions

// and returns the Last matching item from the list

Employee lastMatchEmp = listEmployees.FindLast**(**employee =**>** employee.Gender == "Male"**)**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

lastMatchEmp.ID, lastMatchEmp.Name, lastMatchEmp.Gender, lastMatchEmp.Salary**)**;

Console.WriteLine**(**""**)**;

// Use FindAll() method when you want to return all the items that

// matches the conditions

List**<**Employee**>** filteredEmployees = listEmployees.FindAll**(**employee =**>** employee.Gender == "Male"**)**;

**foreach** **(**Employee femp in filteredEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

femp.ID, femp.Name, femp.Gender, femp.Salary**)**;

**}**

Console.WriteLine**(**""**)**;

// Use FindIndex() method when you want to return the index of the first item

// by a condition

Console.WriteLine**(**"Index of the first matching employee object whose Gender is Male = " +

listEmployees.FindIndex**(**employee =**>** employee.Gender == "Male"**))**;

Console.WriteLine**(**""**)**;

// Use FindLastIndex() method when you want to return the index of the last item

// by a condition

Console.WriteLine**(**"Index of the Last matching employee object whose Gender is Male = " +

listEmployees.FindLastIndex**(**employee =**>** employee.Gender == "Male"**))**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

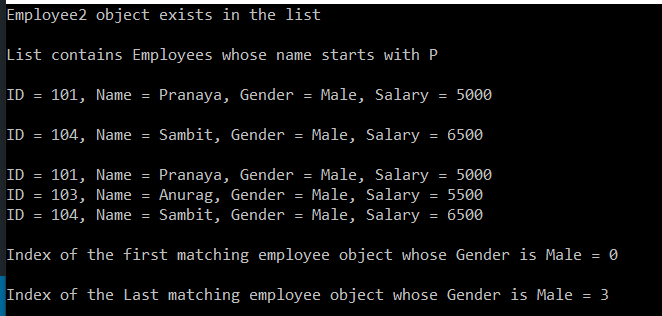
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**Generic List Class Range Methods in C#**

**Generic List Class Range Methods in C#**

In this article, I am going to discuss the **Generic List class Range Methods in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the basics of the [**Generic List class in C#**](https://dotnettutorials.net/lesson/list-collection-csharp/) with examples. The Generic List collection class provides some useful range methods. You can use those range methods to insert or remove a range of elements or items from a list.

**The Generic List class provides the following Range methods.**

**AddRange() Method:**

As we already discussed the Add() method of the List class allows us to add only one item at the end of the collection. If you want to add another list of items to the list collection then you need to the AddRange() method.  
**Syntax: AddRange(IEnumerable<T>)**

**GetRange() Method:**

In our previous article, we discussed that by using the index, we can retrieve only one element from the collection. In many real-time scenarios, we may need to retrieve a list of items from a collection. Then in such scenarios, we need to use the GetRange() method of the List class. The GetRange() method takes 2 parameters. The first parameter is the starting index position and the second parameter is the number of items to return from the list.  
**Syntax: GetRange(Int32, Int32)**

**InsertRange() Method:**

The Insert() method of the Generic List collection class allows us to insert an element at a specified index position. If you want to insert another list of elements at a specified index, then you need to use the InsertRange() method of the List class. This method takes two parameters. The first parameter is the index position where it will insert the elements and the second parameter is the list of items that you want to insert into the collection.  
**Syntax: InsertRange(Int32, IEnumerable<T>)**

**RemoveRange() Method:**

This **RemoveAt**takes the index position of the elements and then removes that element from the collection. If you want to remove a range of elements from a specified index position then you need to use the RemoveRange() method. This RemoveRange() method takes 2 parameters. The first parameter is the start index in the list and the second parameter is the number of elements to remove from the list.  
**Syntax: RemoveRange(Int32, Int32)**

**Note:** The Remove method is used to remove only the first matching element from the list.

**Clear() Method:**

The Clear() method of the List class is used to remove all the items from the list collection.  
**Syntax: Clear()**

**Example: Understanding Range Methods of List Class in C#**

Let us see an example to understand the Generic List class Range Methods in C#. The following code example is self-explained. So please go through the comments.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *ListCollectionrangeMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 5500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Hina",

Gender = "Female",

Salary = 7500

**}**;

Employee emp6 = new Employee**()**

**{**

ID = 106,

Name = "Tarun",

Gender = "Male",

Salary = 8500

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

Console.WriteLine**(**"Add and AddRange Method"**)**;

//Use Add() method to add one item at a time to the end of the list

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

//Create another list

List**<**Employee**>** AnotherlistEmployees = new List**<**Employee**>()**;

AnotherlistEmployees.Add**(**emp4**)**;

AnotherlistEmployees.Add**(**emp5**)**;

AnotherlistEmployees.Add**(**emp6**)**;

//Use AddRange() method to add another list of items

listEmployees.AddRange**(**AnotherlistEmployees**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"GetRange Method"**)**;

//Use GetRange() method to returns a range of items from the list.

List**<**Employee**>** ListOfNewEmployees = listEmployees.GetRange**(**3, 3**)**;

**foreach** **(**Employee employee in ListOfNewEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Remove RemoveAt and RemoveAll"**)**;

//Use Remove() method to removes only the first matching item from the list.

listEmployees.Remove**(**emp1**)**;

//Use RemoveAt() method to remove an item from a specified index.

// listEmployees.RemoveAt(0);

//Use RemoveAll() methid to removes all the items from a collection

// that matches the specified condition.

listEmployees.RemoveAll**(**x =**>** x.Gender == "Female"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"RemoveRange Method"**)**;

// Use RemoveRange() method to removes a range of elements from the list.

listEmployees.RemoveRange**(**0, 2**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Insert and InsertRange Method"**)**;

//Use Insert() method to insert a single item at a specfic position

listEmployees.Insert**(**0, emp1**)**;

listEmployees.Insert**(**1, emp2**)**;

//Use InsertRange() method to insert another list of items at a specified position

listEmployees.InsertRange**(**0, AnotherlistEmployees**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Clear Method"**)**;

// Use Clear method to remove all the items from the list collection

listEmployees.Clear**()**;

Console.WriteLine**(**"Total Items in the List After Clear function = " + listEmployees.Count**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

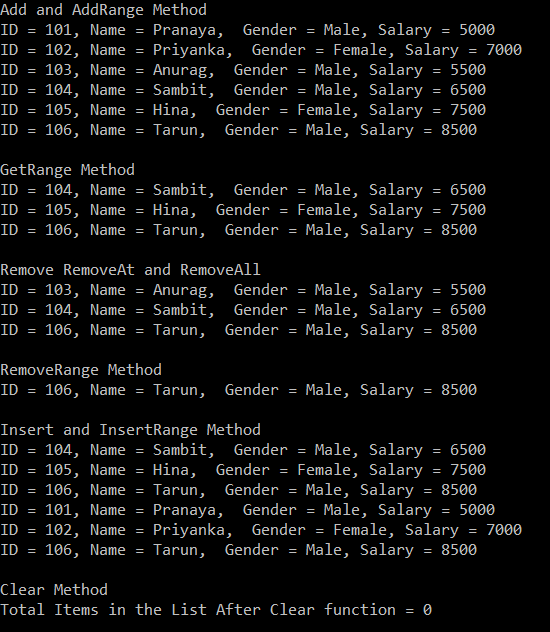
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**How to Sort a List of Simple Type in C#**

**How to Sort a List of Simple Type in C#**

In this article, I am going to discuss **How to Sort a List of Simple Types in C#** with Examples. Please read our previous article where we discussed the [**Range Methods of Generic List in C#**](https://dotnettutorials.net/lesson/generic-list-class-range-methods/) class with examples.

**How to Sort a List of Simple Types in C#?**

In C#, sorting a list of simple types like int, double, char, string, etc. is straightforward. Here, we just need to call the Sort() method which is provided by the Generic List class on the list instance, and then the data will be automatically sorted in ascending order. For example, if we have a list of integers as shown below.  
**List<int> numbersList = new List<int>{ 1, 8, 7, 5, 2, 3, 4, 9, 6 };**  
Then we just need to invoke the Sort() method on numbers list collection as shown below  
**numbersList.Sort();**  
If you want the data to be retrieved in descending order, then use the Reverse() method on the list instance as shown below.  
**numbersList.Reverse();**

However, when we do the same thing on a complex type like Employee, Product, Customer, Department, etc. we get a runtime exception as “**invalid operation exception – Failed to compare 2 elements in the array**”. This is because at runtime the .NET Framework does not identify how to sort the complex types. So, if we want to sort a complex type then we need to tell the way we want the data to be sorted in the list, and to do this we need to implement the **IComparable** interface. We will discuss this in the [**next**](https://dotnettutorials.net/lesson/sorting-a-list-of-complex-type-csharp/)article.

**How the sort functionality is working for simple data types like int, double, string, char, etc. in C#?**

This is working because these types (int, double, string, decimal, char, etc.) are already implementing the **IComparable** interface. If you go to the definition of any built-in types, then you will see that the class is implemented the **IComparable** interface. Let’s see an example to understand this concept.

**namespace** *ListCollectionSortReverseMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<int>** numbersList = new List**<int>** **{** 1, 8, 7, 5, 2, 3, 4, 9, 6 **}**;

Console.WriteLine**(**"Numbers before sorting"**)**;

**foreach** **(int** i in numbersList**)**

**{**

Console.WriteLine**(**i**)**;

**}**

// The Sort() of List Collection class

// will sort the data in ascending order

numbersList.Sort**()**;

Console.WriteLine**(**"Numbers after sorting"**)**;

**foreach** **(int** i in numbersList**)**

**{**

Console.WriteLine**(**i**)**;

**}**

// If you want to to retrieve data in descending order then use the

//Reverse() method of the List collection class

numbersList.Reverse**()**;

Console.WriteLine**(**"Numbers in descending order"**)**;

**foreach** **(int** i in numbersList**)**

**{**

Console.WriteLine**(**i**)**;

**}**

//Another example of sorting alphabets

List**<**string**>** alphabets = new List**<**string**>()** **{** "B", "F", "P", "D", "E", "Z", "A", "C", "L" **}**;

Console.WriteLine**(**"Alphabets before sorting"**)**;

**foreach** **(**string alphabet in alphabets**)**

**{**

Console.WriteLine**(**alphabet**)**;

**}**

alphabets.Sort**()**;

Console.WriteLine**(**"Alphabets after sorting"**)**;

**foreach** **(**string alphabet in alphabets**)**

**{**

Console.WriteLine**(**alphabet**)**;

**}**

alphabets.Reverse**()**;

Console.WriteLine**(**"Alpabets in descending order"**)**;

**foreach** **(**string alphabet in alphabets**)**

**{**

Console.WriteLine**(**alphabet**)**;

**}**

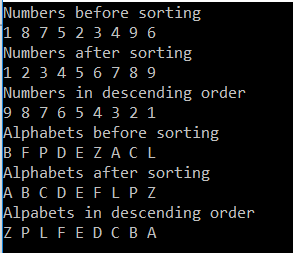
Console.ReadKey**()**;

**}**

**}**

**}**

When we run the application, we will get the following output.



**How to Sort a List of Complex Type in C#**

**How to Sort a List of Complex Type in C#**

In this article, I am going to discuss **how to sort a list of Complex Types in C#** like Employee, Customer, Product, Department, etc. with examples. Please read our previous article before proceeding to this article where we discussed [**how to sort a list of simple types in C#**](https://dotnettutorials.net/lesson/sorting-a-list-of-simple-type-csharp/) like int, double, decimal, string, etc. with an example.

In our last article, we discussed that sorting a list of simple types like int, double, char, string, etc. is straightforward. This means we just need to invoke the Sort() method (Provided by the Generic List class) on the list instance and the data will be automatically sorted in ascending order. For example, if we have a list of integers as shown below  
**List<int> numbersList = new List<int>{ 1, 8, 7, 5, 2, 3, 4, 9, 6 };**  
Then we need to invoke the Sort() method on numbers list collection as shown below  
**numbersList.Sort();**  
However, when we do the same thing on a complex type like Employee, Product, Customer, Department, etc. we get a runtime exception as “**invalid operation exception – Failed to compare 2 elements in the array**”.

**Example: Sorting a List of Complex Type in C#**

In the following example, we will not get any Compile Time Error. But the Invoking **Sort()** on a list of complex types will throw runtime exception i.e. **invalid operation exception** unless we implement the IComparable interface.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *ListCollectionSortReverseMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 4500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Hina",

Gender = "Female",

Salary = 7500

**}**;

Employee emp6 = new Employee**()**

**{**

ID = 106,

Name = "Tarun",

Gender = "Male",

Salary = 6000

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

listEmployees.Add**(**emp5**)**;

listEmployees.Add**(**emp6**)**;

Console.WriteLine**(**"Employees before sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

listEmployees.Sort**()**;

Console.WriteLine**(**"Employees After sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

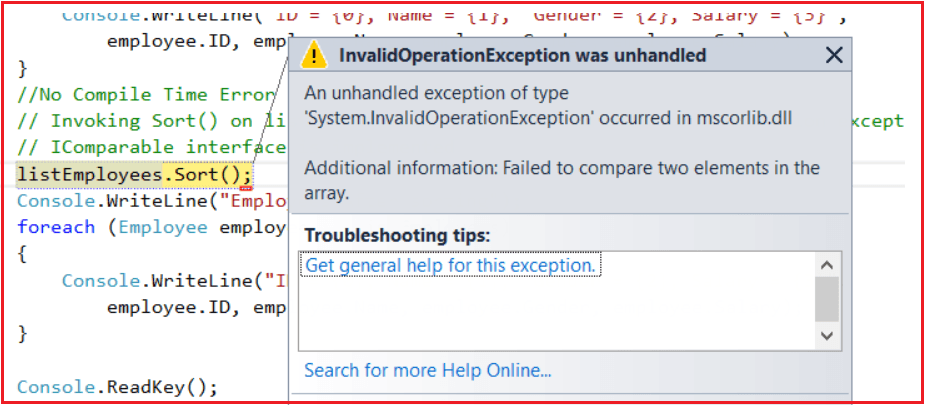
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

When we execute the above code, it will give us the below runtime exception.



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. 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 integer type that we already discussed in our previous article, so we can invoke this method and return it’s valued as shown below.  
**return this.Salary.CompareTo(obj.Salary);**

**Example: Implementing the IComparable interface in C#**

Let see an example for better understanding. What we want is, we need to sort the employees based on the Salary. To do so, our Employee class should implement the **IComparable**interface and should provide an implementation for the **CompareTo()** method. This method will compare the current object (specified with this) and the object to be compared. The following code exactly does the same.

**public** **class** Employee : IComparable**<**Employee**>**

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** **int** CompareTo**(**Employee obj**)**

**{**

**if** **(**this.Salary **>** obj.Salary**)**

**{**

**return** 1;

**}**

**else** **if** **(**this.Salary **<** obj.Salary**)**

**{**

**return** -1;

**}**

**else**

**{**

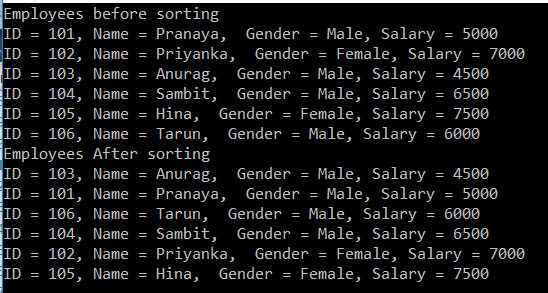
**return** 0;

**}**

**}**

**}**

Now run the application. It will give you the result in ascending order based on the Employee Salary as shown in the below image.



If you prefer not to use the Sort functionality provided by the Employee class, then you can provide your own implementation by implementing the **IComparer** interface. For example, if you want the employees to be sorted by **name** instead of **salary** then you need to follow the below two steps.

**Step1: Implement the IComparer interface**

**public** **class** SortByName : IComparer**<**Employee**>**

**{**

**public** **int** Compare**(**Employee x, Employee y**)**

**{**

**return** x.Name.CompareTo**(**y.Name**)**;

**}**

**}**

**Step2: Pass an instance of the class that implements the IComparer interface as an argument to the Sort() method as shown below.**

**SortByName sortByName = new SortByName();**  
**listEmployees.Sort(sortByName);**

**Here is the complete code:**

Following are the SortByName and Employee classes:

**public** **class** SortByName : IComparer**<**Employee**>**

**{**

**public** **int** Compare**(**Employee x, Employee y**)**

**{**

**return** x.Name.CompareTo**(**y.Name**)**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**Following is the Program class:**

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *ListCollectionSortReverseMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 4500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Hina",

Gender = "Female",

Salary = 7500

**}**;

Employee emp6 = new Employee**()**

**{**

ID = 106,

Name = "Tarun",

Gender = "Male",

Salary = 6000

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

listEmployees.Add**(**emp5**)**;

listEmployees.Add**(**emp6**)**;

Console.WriteLine**(**"Employees before sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

SortByName sortByName = new SortByName**()**;

listEmployees.Sort**(**sortByName**)**;

Console.WriteLine**(**"Employees After sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

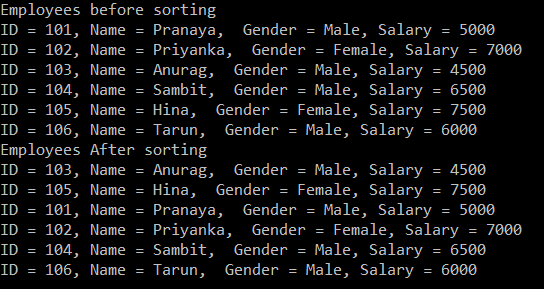
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and should give the following output.



**Comparison Delegate in C#**

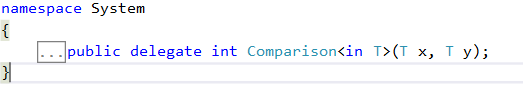
**Comparison delegate in C#**

In this article, I am going to discuss how to sort a list of the complex type using **Comparison Delegate in C#.**Please read our previous article where we discussed [**how to sort a list of Complex Type in C#**](https://dotnettutorials.net/lesson/sorting-a-list-of-complex-type-csharp/)**.**

One of the overload version of the Sort() method in List class expects Comparison delegate to be passed as an argument. Let us understand this overloaded version. The syntax is shown below.

**public void Sort(Comparison<T> comparison)**

**Below is the definition of Comparison Delegate.**



**Approach1:**

**Step1:** Create a function whose signature must match with the signature of Comparison delegate. This is the method where we need to write the logic to compare 2 employee objects.

**private** **static** **int** CompareEmployees**(**Employee e1, Employee e2**)**

**{**

**return** **return** e1.Gender.CompareTo**(**e2.Gender**)**;

**}**

**Step2:**Create an instance of Comparison delegate, and then pass the name of the function created in Step1 as the argument. So, at this point “Comparison” delegate is pointing to our function that contains the logic to compare 2 employee objects.

**Comparison<Employee> employeeComparer= new Comparison<Employee>(CompareEmployees);**

**Step3: Pass the delegate instance as an argument, to Sort() method.**

**listEmployees.Sort(employeeComparer);**

At this point, listEmployees should be sorted using the logic defined in CompareEmployees() function. Below is the complete code:

**namespace** *ListCollectionSortReverseMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 4500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Hina",

Gender = "Female",

Salary = 7500

**}**;

Employee emp6 = new Employee**()**

**{**

ID = 106,

Name = "Tarun",

Gender = "Male",

Salary = 6000

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

listEmployees.Add**(**emp5**)**;

listEmployees.Add**(**emp6**)**;

Console.WriteLine**(**"Employees before sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Comparison**<**Employee**>** employeeComparer = new Comparison**<**Employee**>(**CompareEmployees**)**;

listEmployees.Sort**(**employeeComparer**)**;

Console.WriteLine**(**"Employees After sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** **int** CompareEmployees**(**Employee e1, Employee e2**)**

**{**

**return** e1.Gender.CompareTo**(**e2.Gender**)**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

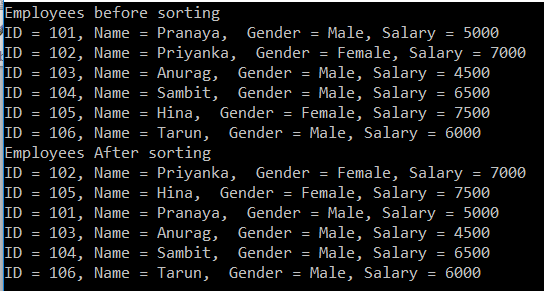
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**



**Approach2:**

In Approach1 this is what we have done

1. We have Created a private function that contains the logic to compare the employees
2. Then created an instance of Comparison delegate, and then passed the name of the private function to the delegate.
3. Finally passed the delegate instance to the Sort() method.

**Do we really have to follow all these steps? Isn’t there any other way?**

The above code can be simplified using the delegate keyword as shown below which is also known as an anonymous method.

listEmployees.Sort**(delegate** **(**Employee e1, Employee e2**)**

**{**

**return** e1.Gender.CompareTo**(**e2.Gender**)**;

**})**;

**Approach3:**

The code in Approach2 can be further simplified using the lambda expression as shown below.

**listCutomers.Sort((x, y) => x.ID.CompareTo(y.ID));**

**The complete code for Approach2 and 3 is given below.**

**namespace** *ListCollectionSortReverseMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 4500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Hina",

Gender = "Female",

Salary = 7500

**}**;

Employee emp6 = new Employee**()**

**{**

ID = 106,

Name = "Tarun",

Gender = "Male",

Salary = 6000

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>()**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

listEmployees.Add**(**emp5**)**;

listEmployees.Add**(**emp6**)**;

Console.WriteLine**(**"Employees before sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

//Apprach2

//Using Anonymous methid

//listEmployees.Sort(delegate (Employee e1, Employee e2)

// {

// return e1.Gender.CompareTo(e2.Gender);

// });

//Approach3

//using Lambda Expression

listEmployees.Sort**((**x, y**)** =**>** x.Gender.CompareTo**(**y.Gender**))**;

Console.WriteLine**(**"Employees After sorting"**)**;

**foreach** **(**Employee employee in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee.ID, employee.Name, employee.Gender, employee.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

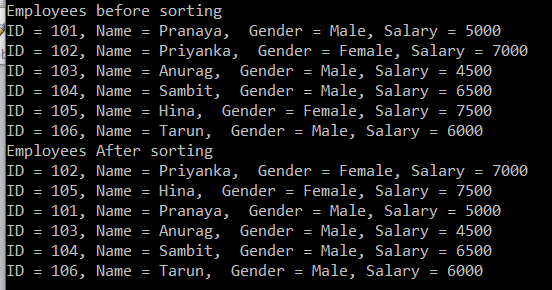
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**



**List Class important methods in C#**

**Generic List Class important methods in C#**

In this article, we will discuss some Generic List Class important methods.

**TrueForAll():**

This method returns true or false depending on whether if every element in the list matches the conditions defined by the specified predicate.

**AsReadOnly():**

This method returns a read-only wrapper for the current collection. Use this method, if you don’t want the client to modify the collection i.e. add or remove any elements from the collection. The ReadOnlyCollection will not have methods to add or remove items from the collection. We can only read items from this collection.

**TrimExcess():**

This method sets the capacity to the actual number of elements in the List if that number is less than a threshold value.

**According to MSDN:**

This method can be used to minimize a collection’s memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large List<T> can be considerable, however, so the TrimExcess method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain. The current threshold is 90 percent, but this could change in the future.

**Let us understand the above methods with an example.**

**namespace** *ListCollectionMethodDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 5000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 7000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 4500

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Male",

Salary = 6500

**}**;

List**<**Employee**>** listEmployees = new List**<**Employee**>(**100**)**;

listEmployees.Add**(**emp1**)**;

listEmployees.Add**(**emp2**)**;

listEmployees.Add**(**emp3**)**;

listEmployees.Add**(**emp4**)**;

//TrueForAll

Console.WriteLine**(**"Are all salaries greater than 5000: "

+ listEmployees.TrueForAll**(**x =**>** x.Salary **>** 5000**))**;

// ReadOnlyCollection will not have Add() or Remove() methods

System.Collections.ObjectModel.ReadOnlyCollection**<**Employee**>**

readOnlyEmployees = listEmployees.AsReadOnly**()**;

Console.WriteLine**(**"Total Items in ReadOnlyCollection = " +

readOnlyEmployees.Count**)**;

// listEmployees list is created with an initial capacity of 50

// but only 4 items are in the list. The filled percentage is

// less than 90 percent threshold.

Console.WriteLine**(**"List capacity before invoking TrimExcess = " +

listEmployees.Capacity**)**;

// Invoke TrimExcess() to set the capacity to the actual

// number of elements in the List

listEmployees.TrimExcess**()**;

Console.WriteLine**(**"List capacity After invoking TrimExcess = " +

listEmployees.Capacity**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

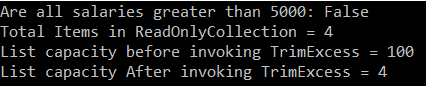
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**



**Dictionary in C#**

**Generic Dictionary in C# with Examples**

In this article, I am going to discuss **Generic Dictionary in C#** with examples. Please read our previous article where we discussed [**Generic List Collection Class**](https://dotnettutorials.net/lesson/list-collection-csharp/) in detail. At the end of this article, you will understand what exactly the Generic Dictionary is and when and how to use Dictionary in C# with examples.

**What is Dictionary in C#?**

The Dictionary in C# is a Collection class same as [**HashTable**](https://dotnettutorials.net/lesson/hashtable-csharp/)i.e. used to store the data in the form of Key-Value Pairs, but here while creating the dictionary object we need to specify the type for the keys as well as the type for values also. The Syntax is given below:

**Dictionary<TKey, TValue> di = new Dictionary<string, object>();**

**Example to Understand Dictionary Collection class:**

Let us understand the Generic Dictionary Collection class in C# with an example. Modify the Program class as shown below. As you can see in the below code, here, we created a dictionary object by specifying the key type as string and value type as an object.

**namespace** *DictionaryCollectionDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Dictionary represents a collection of keys and values

Dictionary**<**string, **object>** di = new Dictionary**<**string, **object>()**;

//adds the specified key and value to the dictionary

di.Add**(**"Eno", 1001**)**;

di.Add**(**"Ename", "Pranaya"**)**;

di.Add**(**"Job", "Developer"**)**;

di.Add**(**"Salary", 7500**)**;

di.Add**(**"Location", "Mumbai"**)**;

//Loopig through each keys to get the values

**foreach** **(**string key in di.Keys**)**

**{**

Console.WriteLine**(**key + " : " + di**[**key**])**;

**}**

Console.WriteLine**()**;

//removes the specified key with the value from the dictionary

Console.WriteLine**(**"After Removing the Job Key : "**)**;

di.Remove**(**"Job"**)**;

**foreach** **(**string key in di.Keys**)**

**{**

Console.WriteLine**(**key + " : " + di**[**key**])**;

**}**

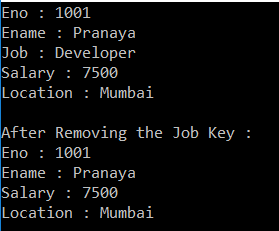
Console.ReadKey**()**;

**}**

**}**

**}**

**Output**:



**Points to Remember while working with Dictionary Generic Collection:**

1. A dictionary is a collection of **key-value** pairs.
2. The Dictionary Generic Collection class is present in **System.Collections.Generic** namespace.
3. When creating a dictionary, we need to specify the **type for the key** and as well as **type for the value**.
4. The fastest way to find a value in a dictionary is by using the keys.
5. Keys in a dictionary must be unique.

**Example: Dictionary Key as Integer and Value as Complex Type in C#**

Let’s use see an example of dictionary generic collection using the key as an integer and values as a complex type. The below example code is self-explained. Please go through the comment lines.

**namespace** *DictionaryCollectionDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create Employee object

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 20000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 30000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 40000

**}**;

// Create a Dictionary collection where

// Employee ID is the key and the key Type is int

// Employee object is the value and the value Type is Employee

Dictionary**<int**, Employee**>** dictionaryEmployees = new Dictionary**<int**, Employee**>()**;

// Add Employee objects to the dictionary collection

// Employee ID is the key and the employee object is the value

dictionaryEmployees.Add**(**emp1.ID, emp1**)**;

dictionaryEmployees.Add**(**emp2.ID, emp2**)**;

dictionaryEmployees.Add**(**emp3.ID, emp3**)**;

// Retrieve the value (i.e. Employee object) from the dictionary

// using the key (i.e. Employee ID).

// The fastest way to get a value from the dictionary is by using its key

Employee employee101 = dictionaryEmployees**[**101**]**;

Console.WriteLine**(**"Employee 101 in employee dictionary"**)**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender = {2}, Salary = {3}",

employee101.ID, employee101.Name, employee101.Gender, employee101.Salary**)**;

Console.WriteLine**()**;

// It is also possible to loop thru each key/value pair in a dictionary

Console.WriteLine**(**"All Employees keys and values in employee dictionary"**)**;

**foreach** **(**KeyValuePair**<int**, Employee**>** employeeKeyValuePair in dictionaryEmployees**)**

**{**

Console.WriteLine**(**"Key = " + employeeKeyValuePair.Key**)**;

Employee emp = employeeKeyValuePair.Value;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// We can also use implicitly typed variable var to loop thru

// each key/value pair in a dictionary.

// But try to avoid using var, as this makes our code less readable

Console.WriteLine**(**"All Employees keys and values in employee dictionary"**)**;

**foreach** **(**var employeeKeyValuePair in dictionaryEmployees**)**

**{**

Console.WriteLine**(**"Key = " + employeeKeyValuePair.Key**)**;

Employee emp = employeeKeyValuePair.Value;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// To get all the keys in the dictionary we have to use the keys properties

// of dictionaryCustomers object as shown below

Console.WriteLine**(**"All Keys in Employee Dictionary"**)**;

**foreach** **(int** key in dictionaryEmployees.Keys**)**

**{**

Console.WriteLine**(**key + " "**)**;

**}**

Console.WriteLine**()**;

// First get the keys, then get the values using the keys

Console.WriteLine**(**"All Keys and values in Employee Dictionary"**)**;

**foreach** **(int** key in dictionaryEmployees.Keys**)**

**{**

Console.WriteLine**(**key + " "**)**;

Employee emp = dictionaryEmployees**[**key**]**;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

//To get all the values in the dictionary use Values property

Console.WriteLine**(**"All employees objects in Employee Dictionary"**)**;

**foreach** **(**Employee emp in dictionaryEmployees.Values**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

// If we try to add a key that already exists in the dictionary we will get an exception -

// An item with the same key has already been added. So, check if the key already exists

**if** **(**!dictionaryEmployees.ContainsKey**(**101**))**

**{**

dictionaryEmployees.Add**(**101, emp1**)**;

**}**

Console.WriteLine**()**;

// When accessing a dictionary value by key, make sure the dictionary contains the key,

// otherwise we will get KeyNotFound exception.

**if** **(**dictionaryEmployees.ContainsKey**(**110**))**

**{**

Employee emp = dictionaryEmployees**[**110**]**;

**}**

**else**

**{**

Console.WriteLine**(**"Key does not exist in the dictionary"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

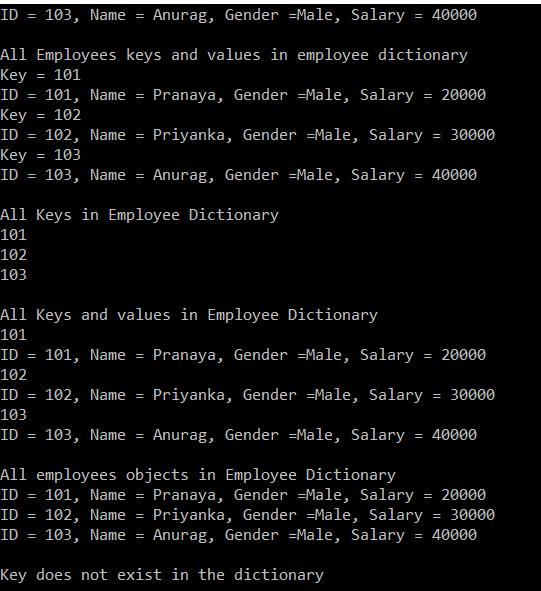
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**Collection Initializers in C#:**

This is a new feature added to C# 3.0 which allows initializing a collection directly at the time of declaration like an array. A Dictionary<TKey,TValue> contains a collection of key/value pairs. Its Add method takes two parameters, one for the key and one for the value. To initialize a **Dictionary<TKey, TValue>**, or any collection whose Add method takes multiple parameters, enclose each set of parameters in braces.

**Example:**

**namespace** *DictionaryCollectionDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

// Collection initializer

// Initializing the collection directly at the time of declaration

Dictionary**<int**, Employee**>** dictionaryEmployees = new Dictionary**<int**, Employee**>()**

**{**

**{** 101, new Employee **{**ID=101, Name="Pranaya", Gender="Male", Salary = 20000**}}**,

**{** 102, new Employee **{**ID=101, Name="Priyanka", Gender="Female", Salary = 30000**}}**,

**{** 103, new Employee **{**ID=101, Name="Anurag", Gender="Male", Salary = 40000**}}**

**}**;

//To get all the values in the dictionary use Values property

Console.WriteLine**(**"All employees objects in Employee Dictionary"**)**;

**foreach** **(**Employee emp in dictionaryEmployees.Values**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

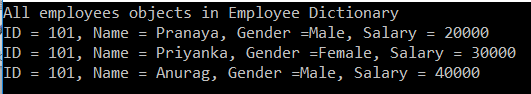
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**

:

**Dictionary Class important Methods in C#:**

The Generic Dictionary Collection class stores the data in the form of key-value pairs. Now, let us see the following Dictionary class methods in detail.

**TryGetValue():**This method takes two parameters, one is the key and the other one is the value. The value type parameter is of type out parameter. If the key exists in the dictionary then it will return true and the value with that associated key is stored on the output variable. If you are not sure if a key is present or not in the dictionary, then you can use the TryGetValue() method to get the value from a dictionary because if you are not using TryGetValue then at that case you will get KeyNotFoundException

**Count():**The Count() function is used to find the total number of items in a dictionary.

**Remove():**If you want to remove an item from the dictionary collection, then you need to use the Remove() method.

**Clear():** The Clear() method is used to remove all the items from the dictionary.

**Note:** We will see how to use LINQ extension methods with Dictionary and different ways to convert an array into a dictionary.

**Example to understand the above methods.**

The code is self-explained. Please go through the comments.

**namespace** *DictionaryCollectionDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create Employee object

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 20000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 30000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 40000

**}**;

// Create a Dictionary collection where

// Employee ID is the key and the key Type is int

// Employee object is the value and the value Type is Employee

Dictionary**<int**, Employee**>** dictionaryEmployees = new Dictionary**<int**, Employee**>()**;

// Add Employee objects to the dictionary collection

// Employee ID is the key and the employee object is the value

dictionaryEmployees.Add**(**emp1.ID, emp1**)**;

dictionaryEmployees.Add**(**emp2.ID, emp2**)**;

dictionaryEmployees.Add**(**emp3.ID, emp3**)**;

// If you are not sure if a key is present or not in the dictionary,

// then you can use the TryGetValue() method to get the value from a dictionary because

// if you are not using TryGetValue then at that case you will get KeyNotFoundException

Employee emp777;

**if** **(**dictionaryEmployees.TryGetValue**(**777, out emp777**))**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp777.ID, emp777.Name, emp777.Gender, emp777.Salary**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Employee with Key = 777 is not found in the dictionary"**)**;

Console.WriteLine**()**;

**}**

// To find the total number of items in a dictionary use Count() method

Console.WriteLine**(**"Total items in Employee Dictionary = {0}", dictionaryEmployees.Count**())**;

// Or You can use the Count Property

// Console.WriteLine("Total items in Employee Dictionary = {0}", dictionaryEmployees.Count);

Console.WriteLine**()**;

// LINQ extension methods can also be used with the Dictionary.

// For example, to find the total employees whose Gender is Male.

Console.WriteLine**(**"Total Employees in the dictionary where Gender is Male = {0}",

dictionaryEmployees.Count**(**kvp =**>** kvp.Value.Gender == "Male"**))**;

Console.WriteLine**()**;

// To remove an item from the dictionary, use Remove() method

dictionaryEmployees.Remove**(**101**)**;

// To remove all items from the dictionary, use Clear() method

dictionaryEmployees.Clear**()**;

//Now our dictionary is empty

// Create an array of employees with size 3

// Store the 3 employees into the array

Employee**[]** arrayEmployees = new Employee**[**3**]**;

arrayEmployees**[**0**]** = emp1;

arrayEmployees**[**1**]** = emp2;

arrayEmployees**[**2**]** = emp3;

// Convert Employee array to a dictionary using ToDictionary() method.

// In this example, key is Employee ID and the value is the employee object itself

Dictionary**<int**, Employee**>** dict = arrayEmployees.ToDictionary**(**employee =**>** employee.ID, employee =**>** employee**)**;

// OR

// Dictionary<int, Employee> dict = arrayEmployees.ToDictionary(employee => employee.ID);

//OR use a foreach loop

//Dictionary< int, Employee> dict = new Dictionary<int, Employee>();

//foreach (Employee emp in arrayEmployees)

//{

// dict.Add(emp.ID, emp);

//}

// Loop thru the dictionary and print the key/value pairs

**foreach** **(**KeyValuePair**<int**, Employee**>** kvp in dict**)**

**{**

Console.WriteLine**(**"Key = {0}", kvp.Key**)**;

Employee emp = kvp.Value;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

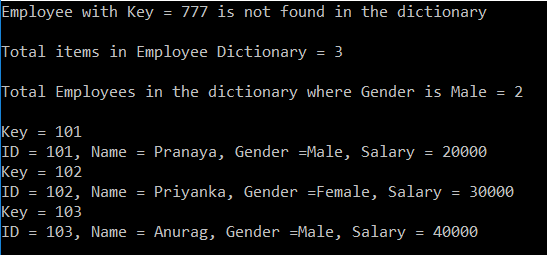
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**Conversion Between Array List and Dictionary in C#**

**Conversion between Array List and Dictionary in C#**

In this article, we will discuss how to perform **Conversion Between Array List and Dictionary in C#**. Please read our previous article where we discussed [**Dictionary in C#**](https://dotnettutorials.net/lesson/dictionary-generic-collection-csharp/) with examples. As part of this article, we will discuss the following six things.

1. **Convert an array to a List – Use ToList() method**
2. **Convert a list to an array – Use ToArray() method**
3. **Convert a List to a Dictionary – Use ToDictionary() method**
4. **Convert an array to a Dictionary – Use ToDictionary() method**
5. **Convert a Dictionary to an array – Use ToArray() method on the Values Property of the dictionary object**
6. **Convert a Dictionary to a List – Use the ToList() method on the Values Property of the dictionary object**

**Let us understand this with an example.**

The code is self-explained. Please go through the comments.

**namespace** *DictionaryCollectionDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create Employee object

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 20000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 30000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 40000

**}**;

// Create an array of employees with size 3

// Store the 3 employees into the array

Employee**[]** arrayEmployees = new Employee**[**3**]**;

arrayEmployees**[**0**]** = emp1;

arrayEmployees**[**1**]** = emp2;

arrayEmployees**[**2**]** = emp3;

// To convert an array to a List, use ToList() method

List**<**Employee**>** listEmployees = arrayEmployees.ToList**()**;

**foreach** **(**Employee emp in listEmployees**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// To convert a List to an array, use ToLArray() method

Employee**[]** arrayAllEmployeesFromList = listEmployees.ToArray**()**;

**foreach** **(**Employee emp in arrayAllEmployeesFromList**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// To convert a List to a Dictionary, use ToDictionary() method

Dictionary**<int**, Employee**>** dictionaryEmployees = listEmployees.ToDictionary**(**x =**>** x.ID**)**;

**foreach** **(**KeyValuePair**<int**, Employee**>** keyValuePairEmployees in dictionaryEmployees**)**

**{**

Console.WriteLine**(**"Key = {0}", keyValuePairEmployees.Key**)**;

Employee emp = keyValuePairEmployees.Value;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// To convert an array to a Dictionary, use ToDictionary() method

Dictionary**<int**, Employee**>** dictionaryEmployeesFromArray = arrayEmployees.ToDictionary**(**employee =**>** employee.ID, employee =**>** employee**)**;

// Loop thru the dictionary and print the key/value pairs

**foreach** **(**KeyValuePair**<int**, Employee**>** kvp in dictionaryEmployeesFromArray**)**

**{**

Console.WriteLine**(**"Key = {0}", kvp.Key**)**;

Employee emp = kvp.Value;

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

// To Convert a dictionaty to an array, use ToArray method on the Values

// Peoperty of the dictionary object

Employee**[]** arrayAllEmployeesFromDictionary = dictionaryEmployeesFromArray.Values.ToArray**()**;

**foreach** **(**Employee emp in arrayAllEmployeesFromDictionary**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.WriteLine**()**;

// To Convert a dictionary to a List, use To List method on the Values

// Property of the dictionary object

List**<**Employee**>** listAllEmployeesFromDictionary = dictionaryEmployeesFromArray.Values.ToList**()**;

**foreach** **(**Employee emp in listAllEmployeesFromDictionary**)**

**{**

Console.WriteLine**(**"ID = {0}, Name = {1}, Gender ={2}, Salary = {3}",

emp.ID, emp.Name, emp.Gender, emp.Salary**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

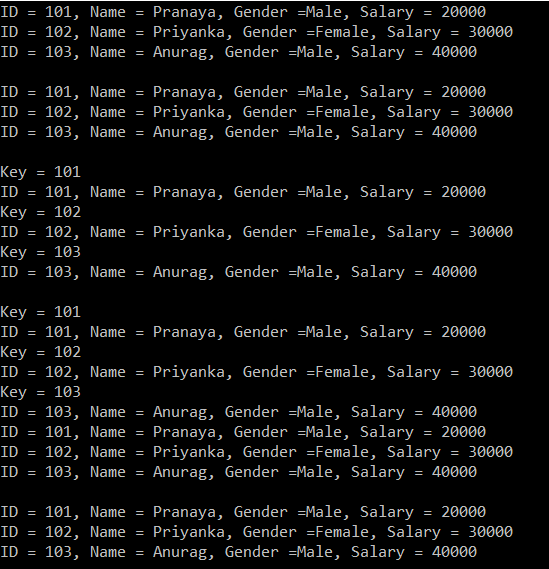
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**List vs Dictionary in C#**

**List vs Dictionary in C# with Examples**

In this article, I am going to discuss **List vs Dictionary in C#** with Examples. Please read our previous article where we discussed [**Conversion Between Array List and Dictionary**](https://dotnettutorials.net/lesson/conversion-between-array-list-and-dictionary-csharp/) in C#. At the end end of this article, you will understand the difference between List and Dictionary as well as you will understand when to use List over Dictionary and vice-versa.

**List vs Dictionary in C#**

Both lists and dictionaries belong to Generics collections that is used to store collections of data. Both **Dictionary <TKey, TValue>** and **List <T>** are similar both have random access data structures on top of the .NET framework. The Dictionary is based on a hash table that means it uses a hash lookup, which is an efficient algorithm to look up things, on the other hand, a list, has to go and check element by element until it finds the result from the beginning. In this article, we will discuss List vs Dictionary in C#. When comparing with the List data structure, the dictionary is always a more or less fixed lookup time.

**Let’s go into the details.**

The Dictionary uses the hashing algorithm to search for the element (data). A Dictionary first calculates a hash value for the key and this hash value leads to the target data bucket. After that, each element in the bucket needs to be checked for equality. But actually, the list will be faster than the dictionary on the first item search because nothing to search in the first step. But in the second step, the list has to look through the first item and then the second item. So each step of the lookup takes more and more time. The larger the list, the longer it takes. Of course, the Dictionary in principle has a faster lookup with O(1) while the lookup performance of a List is an O(n) operation.

The Dictionary maps a key to a value and cannot have duplicate keys, whereas a list just contains a collection of values. Also, Lists allow duplicate items and support linear traversal.

Consider the following example:  
**Dictionary<string, int> dictionary = new Dictionary<string, int>();**  
**List<int> newList = new List<int>();**

Add data to the list  
**newList.Add(data);**

A list can simply add the item at the end of the existing list item. Add data to the Dictionary  
**dictionary.Add(key, data);**

When you add data to a Dictionary, you should specify a unique key to the data so that it can be uniquely identified.

A Dictionary has a unique identifier, so whenever you look up a value in a Dictionary, the runtime must compute a hash code from the key. This optimized algorithm is implemented by some low-level bit shifting or modulo divisions. We determine the point at which Dictionary becomes more efficient for lookups than List.

**Example to understand List vs Dictionary in C#:**

The Find() method of the List class loops thru each object in the list until a match is found. So, if we want to look up a value using a key, then a dictionary is better for performance over the list. So, we need to use a dictionary when we know the collection will be primarily used for lookups.

**namespace** *DictionaryCollectionDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Country country1 = new Country**()**

**{**

Code = "AUS",

Name = "AUSTRALIA",

Capital = "Canberra"

**}**;

Country country2 = new Country**()**

**{**

Code = "IND",

Name = "INDIA ",

Capital = "New Delhi"

**}**;

Country country3 = new Country**()**

**{**

Code = "USA",

Name = "UNITED STATES",

Capital = "Washington D.C."

**}**;

Country country4 = new Country**()**

**{**

Code = "GBR",

Name = "UNITED KINGDOM",

Capital = "London"

**}**;

Country country5 = new Country**()**

**{**

Code = "CAN",

Name = "CANADA",

Capital = "Ottawa"

**}**;

//List<Country> listCountries = new List<Country>();

//listCountries.Add(country1);

//listCountries.Add(country2);

//listCountries.Add(country3);

//listCountries.Add(country4);

//listCountries.Add(country5);

Dictionary**<**string, Country**>** dictionaryCountries = new Dictionary**<**string, Country**>()**;

dictionaryCountries.Add**(**country1.Code, country1**)**;

dictionaryCountries.Add**(**country2.Code, country2**)**;

dictionaryCountries.Add**(**country3.Code, country3**)**;

dictionaryCountries.Add**(**country4.Code, country4**)**;

dictionaryCountries.Add**(**country5.Code, country5**)**;

string strUserChoice = string.Empty;

**do**

**{**

Console.WriteLine**(**"Please enter country code"**)**;

string strCountryCode = Console.ReadLine**()**.ToUpper**()**;

// Find() method of the list class loops thru each object in the list until a match is found. So, if we want to

// lookup a value using a key dictionary is better for performance over list.

// Country resultCountry = listCountries. Find(country => country.Code == strCountryCode);

Country resultCountry = dictionaryCountries.ContainsKey**(**strCountryCode**)** ? dictionaryCountries**[**strCountryCode**]** : **null**;

**if** **(**resultCountry == **null)**

**{**

Console.WriteLine**(**"The country code you entered does not exist"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Name = " + resultCountry.Name + " Captial =" + resultCountry.Capital**)**;

**}**

**do**

**{**

Console.WriteLine**(**"Do you want to continue - YES or NO?"**)**;

strUserChoice = Console.ReadLine**()**.ToUpper**()**;

**}**

**while** **(**strUserChoice != "NO" && strUserChoice != "YES"**)**;

**}**

**while** **(**strUserChoice == "YES"**)**;

// Console.ReadKey();

**}**

**}**

**public** **class** Country

**{**

**public** string Name **{** **get**; **set**; **}**

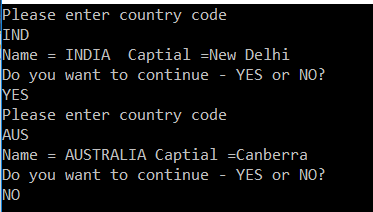
**public** string Code **{** **get**; **set**; **}**

**public** string Capital **{** **get**; **set**; **}**

**}**

**}**

**Output:**



**Generic Stack in C#**

**Generic Stack in C# with Examples**

In this article, I am going to discuss **Generic Stack in C#** with examples. Please read our previous article where we discussed the difference between [**List vs Dictionary**](https://dotnettutorials.net/lesson/dictionary-vs-list-csharp/) in detail. At the end of this article, you will understand what exactly Generic Stack is and when and how to use Generic Stack in C# with examples.

**What is Generic Stack in C#?**

The **Generic Stack** in C# is a collection class which works on the principle of Last In First Out (LIFO) and this class is present in **System.Collections.Generic** namespace.

This Stack collection class is analogous to a stack of plates. For example, if we want to add a new plate to the stack of plates, then we will only place it on top of all the already existing plates. Similarly, if we want to remove a plate from the stack, then we will only remove the one that we have last added. The stack collection class also operates in the similar fashion. The last item to be added (pushed) to the stack will be the first item to be removed (popped) from the stack.

**Methods of Generic Stack Class:**

To insert an item at the top of the stack, we need to use the**Push()** method. Similarly, to remove and return the item that is present at the top of the stack, use**Pop()** method.

We can use a foreach loop to iterate thru all the items of a stack. The items from the stack are retrieved inLIFO(Last In First Out), order. The last element added to the Stack is the first item to be removed. To check if an item exists in the stack, use**Contains()** method.

**What is the difference between Pop() and Peek() methods?**

The Pop() method removes and returns the item at the top of the stack, whereas the Peek() method returns the item from the top of the stack, without removing it.

**Understanding the Generic Stack collection class with an example.**

Let us understand the Generic Stack Collection class with an example. Create one console application and then copy and paste the following code in the Program class. The following code is self-explained, so please go through the comments.

**namespace** *GenericStackDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create Employee object

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 20000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 30000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 40000

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Female",

Salary = 40000

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Preety",

Gender = "Female",

Salary = 50000

**}**;

// Create a Generic Stack of Employees

Stack**<**Employee**>** stackEmployees = new Stack**<**Employee**>()**;

// To add an item into the stack, use the Push() method.

// emp1 is inserted at the top of the stack

stackEmployees.Push**(**emp1**)**;

// emp2 will be inserted on top of emp1 and now is on top of the stack

stackEmployees.Push**(**emp2**)**;

// emp3 will be inserted on top of emp2 and now is on top of the stack

stackEmployees.Push**(**emp3**)**;

// emp4 will be inserted on top of emp3 and now is on top of the stack

stackEmployees.Push**(**emp4**)**;

// emp5 will be inserted on top of emp4 and now is on top of the stack

stackEmployees.Push**(**emp5**)**;

// If you need to loop thru each items in the stack, then we can use the foreach loop

// in the same way as we use it with other collection classes.

// The foreach loop will only iterate thru the items in the stack, but will not remove them.

// Notice that 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.

Console.WriteLine**(**"Retrive Using Foreach Loop"**)**;

**foreach** **(**Employee emp in stackEmployees**)**

**{**

Console.WriteLine**(**emp.ID + " - " + emp.Name + " - " + emp.Gender + " - " + emp.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

**}**

Console.WriteLine**(**"------------------------------"**)**;

// To retrieve an item from the stack, use the Pop() method.

// This method removes and returns an object at the top of the stack.

// Since emp5 object is the one that is pushed onto the stack last, this object will be

// first to be removed and returned from the stack by the Pop() method

Console.WriteLine**(**"Retrive Using Pop Method"**)**;

Employee e1 = stackEmployees.Pop**()**;

Console.WriteLine**(**e1.ID + " - " + e1.Name + " - " + e1.Gender + " - " + e1.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Employee e2 = stackEmployees.Pop**()**;

Console.WriteLine**(**e2.ID + " - " + e2.Name + " - " + e2.Gender + " - " + e2.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Employee e3 = stackEmployees.Pop**()**;

Console.WriteLine**(**e3.ID + " - " + e3.Name + " - " + e3.Gender + " - " + e3.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Employee e4 = stackEmployees.Pop**()**;

Console.WriteLine**(**e4.ID + " - " + e4.Name + " - " + e4.Gender + " - " + e4.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Employee e5 = stackEmployees.Pop**()**;

Console.WriteLine**(**e5.ID + " - " + e5.Name + " - " + e5.Gender + " - " + e5.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Console.WriteLine**(**"------------------------------"**)**;

// Now there will be no items left in the stack.

// So, let's push the five objects once again

stackEmployees.Push**(**emp1**)**;

stackEmployees.Push**(**emp2**)**;

stackEmployees.Push**(**emp3**)**;

stackEmployees.Push**(**emp4**)**;

stackEmployees.Push**(**emp5**)**;

// To retrieve an item that is present at the top of the stack,

// without removing it, then use the Peek() method.

Console.WriteLine**(**"Retrive Using Peek Method"**)**;

Employee e105 = stackEmployees.Peek**()**;

Console.WriteLine**(**e105.ID + " - " + e105.Name + " - " + e105.Gender + " - " + e105.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Employee e104 = stackEmployees.Peek**()**;

Console.WriteLine**(**e104.ID + " - " + e104.Name + " - " + e104.Gender + " - " + e104.Salary**)**;

Console.WriteLine**(**"Items left in the Stack = " + stackEmployees.Count**)**;

Console.WriteLine**(**"------------------------------"**)**;

// To check if an item exists in the stack, use Contains() method.

**if** **(**stackEmployees.Contains**(**emp3**))**

**{**

Console.WriteLine**(**"Emp3 is in stack"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Emp3 is not in stack"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

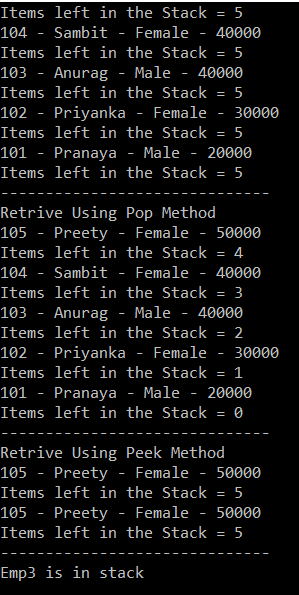
**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**



**Generic Queue in C#**

**Generic Queue in C# with Examples**

In this article, I am going to discuss **Generic Queue in C#** with examples. Please read our previous article where we discussed the [**Generic Stack Collection Class**](https://dotnettutorials.net/lesson/generic-stack-csharp/) in detail. At the end of this article, you will understand what exactly Generic Queue is and when and how to use Generic Queue in C# with examples.

**What is Generic Queue in C#?**

The **Generic Queue** is a collection class which works on the principle of **First In First Out (FIFO)** and this class 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, on 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 also operates in the same fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed (dequeued) from the Queue.

**Methods of Generic Queue class in C#:**

To add items to the end of the queue, use the **Enqueue()** method. Similarly, to remove an item that is present at the beginning of the queue, use **Dequeue()** method. A foreach loop will iterate thru all 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.

**What is the difference between Dequeue() and Peek() methods?**

The Dequeue() method removes and returns the item at the beginning of the queue, whereas the Peek() method returns the item at the beginning of the queue, without removing it.

**Understanding the Generic Queue Collection class with an example.**

Let us understand the Generic Queue Collection class with an example. First, create one console application and then modify the Program class as shown below. The following code is self-explained, so please go through the comments.

**namespace** *GenericQueueDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create Employee object

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Gender = "Male",

Salary = 20000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Priyanka",

Gender = "Female",

Salary = 30000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Anurag",

Gender = "Male",

Salary = 40000

**}**;

Employee emp4 = new Employee**()**

**{**

ID = 104,

Name = "Sambit",

Gender = "Female",

Salary = 40000

**}**;

Employee emp5 = new Employee**()**

**{**

ID = 105,

Name = "Preety",

Gender = "Female",

Salary = 50000

**}**;

// Create a Generic Queue of Employees

Queue**<**Employee**>** queueEmployees = new Queue**<**Employee**>()**;

// To add an item into the queue, use the Enqueue() method.

// emp1 is added first, so this employee, will be the first to get out of the queue

queueEmployees.Enqueue**(**emp1**)**;

// emp2 will be queued up next, so employee 2 will be second to get out of the queue

queueEmployees.Enqueue**(**emp2**)**;

// emp3 will be queued up next, so employee 3 will be third to get out of the queue

queueEmployees.Enqueue**(**emp3**)**;

// emp3 will be queued up next, so employee 4 will be fourth to get out of the queue

queueEmployees.Enqueue**(**emp4**)**;

// emp5 will be queued up next, so employee 5 will be fifth to get out of the queue

queueEmployees.Enqueue**(**emp5**)**;

// If you need to loop thru each items in the queue, then we can use the foreach loop

// in the same way as we use it with other collection classes.

// The foreach loop will only iterate thru the items in the queue, but will not remove them.

// Notice that the items from the queue are retrieved in FIFI (First In First Out), order.

// The First element added to the queue is the first one to be removed.

Console.WriteLine**(**"Retrive Using Foreach Loop"**)**;

**foreach** **(**Employee emp in queueEmployees**)**

**{**

Console.WriteLine**(**emp.ID + " - " + emp.Name + " - " + emp.Gender + " - " + emp.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

**}**

Console.WriteLine**(**"------------------------------"**)**;

// To retrieve an item from the queue, use the Dequeue() method.

// Notice that the items are dequeued in the same order in which they were enqueued.

// Dequeue() method removes and returns the item at the beginning of the Queue.

// Since emp1 object is the one that is enqueued onto the queue first, this object will be

// first to be dequeued and returned from the queue by using Dequeue() method

Console.WriteLine**(**"Retrive Using Dequeue Method"**)**;

Employee e1 = queueEmployees.Dequeue**()**;

Console.WriteLine**(**e1.ID + " - " + e1.Name +" - "+ e1.Gender + " - " + e1.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Employee e2 = queueEmployees.Dequeue**()**;

Console.WriteLine**(**e2.ID + " - " + e2.Name + " - " + e2.Gender + " - " + e2.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Employee e3 = queueEmployees.Dequeue**()**;

Console.WriteLine**(**e3.ID + " - " + e3.Name + " - " + e3.Gender + " - " + e3.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Employee e4 = queueEmployees.Dequeue**()**;

Console.WriteLine**(**e4.ID + " - " + e4.Name + " - " + e4.Gender + " - " + e4.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Employee e5 = queueEmployees.Dequeue**()**;

Console.WriteLine**(**e5.ID + " - " + e5.Name + " - " + e5.Gender + " - " + e5.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Console.WriteLine**(**"------------------------------"**)**;

// Now there will be no items left in the queue.

// So, let's Enqueue the five objects once again

queueEmployees.Enqueue**(**emp1**)**;

queueEmployees.Enqueue**(**emp2**)**;

queueEmployees.Enqueue**(**emp3**)**;

queueEmployees.Enqueue**(**emp4**)**;

queueEmployees.Enqueue**(**emp5**)**;

// To retrieve an item that is present at the beginning of the queue,

// without removing it, then use the Peek() method.

Console.WriteLine**(**"Retrive Using Peek Method"**)**;

Employee e101 = queueEmployees.Peek**()**;

Console.WriteLine**(**e101.ID + " - " + e101.Name + " - " + e101.Gender + " - " + e101.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Employee e103 = queueEmployees.Peek**()**;

Console.WriteLine**(**e103.ID + " - " + e103.Name + " - " + e103.Gender + " - " + e103.Salary**)**;

Console.WriteLine**(**"Items left in the Queue = " + queueEmployees.Count**)**;

Console.WriteLine**(**"------------------------------"**)**;

// To check if an item exists in the stack, use Contains() method.

**if** **(**queueEmployees.Contains**(**emp3**))**

**{**

Console.WriteLine**(**"Emp3 is in Queue"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Emp3 is not in queue"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

**OUTPUT:**

