**Collections**

Generics are used to make reusable code classes to decrease the code redundancy, increase type safety and performance. Using generics, we can create collection classes. To create generic collection, System.Collections.Generic namespace should be used instead of classes such as ArrayList in the System.Collections namespace. Generics promotes the usage of parameterized types.

In this chapter we will deal with C# collections. The .NET framework provides specialized classes for data storage and retrieval. In one of the previous chapters, we have described arrays. Collections are enhancement to the arrays.

There are two distinct collection types in C#. The standard collections, which are found under the System.Collections namespace and the generic collections, under System.Collections.Generic. The generic collections are more flexible and are the preferred way to work with data. The generic collections or generics were introduced in .NET framework 2.0. Generics enhance code reuse, type safety, and performance.

*Generic programming* is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by Ada in 1983, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. (Wikipedia)

**ArrayList**

An ArrayList is a collection from a standard System.Collections namespace. It is a dynamic array. It provides random access to its elements. An ArrayList automatically expands as data is added. Unlike arrays, an ArrayList can hold data of multiple data types. Elements in the ArrayList are accessed via an integer index. Indexes are zero based. Indexing of elements and insertion and deletion at the end of the ArrayList takes constant time. Inserting or deleting an element in the middle of the dynamic array is more costly. It takes linear time.

using System;

using System.Collections;

public class CSharpApp

{

class Empty

{}

static void Main()

{

ArrayList da = new ArrayList();

da.Add("Visual Basic");

da.Add(344);

da.Add(55);

da.Add(new Empty());

da.Remove(55);

foreach(object el in da)

{

Console.WriteLine(el);

}

}

}

In the above example, we have created an ArrayList collection. We have added some elements to it. They are of various data type, string, int and a class object.

using System.Collections;

In order to work with ArrayList collection, we need to use the System.Collections namespace.

ArrayList da = new ArrayList();

An ArrayList collection is created.

da.Add("Visual Basic");

da.Add(344);

da.Add(55);

da.Add(new Empty());

We add four elements to the array with the Add() method.

da.Remove(55);

We remove one element with the Remove() method.

foreach(object el in da)

{

Console.WriteLine(el);

}

We iterate through the array and print its elements to the console.

$ ./arraylist.exe

Visual Basic

344

CSharpApp+Empty

This is the output of the example.

**List**

A List is a strongly typed list of objects that can be accessed by index. It can be found under System.Collections.Generic namespace.

using System;

using System.Collections.Generic;

public class ListExample

{

static void Main()

{

List<string> langs = new List<string>();

langs.Add("Java");

langs.Add("C#");

langs.Add("C");

langs.Add("C++");

langs.Add("Ruby");

langs.Add("Javascript");

Console.WriteLine(langs.Contains("C#"));

Console.WriteLine(langs[1]);

Console.WriteLine(langs[2]);

langs.Remove("C#");

langs.Remove("C");

Console.WriteLine(langs.Contains("C#"));

langs.Insert(4, "Haskell");

langs.Sort();

foreach(string lang in langs)

{

Console.WriteLine(lang);

}

}

}

In the preceding example, we work with the List collection.

using System.Collections.Generic;

The List collection is located in the System.Collections.Generic namespace.

List<string> langs = new List<string>();

A generic dynamic array is created. We specify that we will work with strings with the type specified inside <> characters.

langs.Add("Java");

langs.Add("C#");

langs.Add("C");

...

We add elements to the List using the Add() method.

Console.WriteLine(langs.Contains("C#"));

We check if the List contains a specific string using the Contains() method.

Console.WriteLine(langs[1]);

Console.WriteLine(langs[2]);

We access the second and the third element of the List using the index notation.

langs.Remove("C#");

langs.Remove("C");

We remove two strings from the List.

langs.Insert(4, "Haskell");

We insert a string at a specific location.

langs.Sort();

We sort the elements using the Sort() method.

$ ./list.exe

True

C#

C

False

C++

Haskell

Java

Javascript

Ruby

This is the outcome of the example.

**LinkedList**

LinkedList is a generic doubly linked list in C#. LinkedList only allows sequential access. LinkedList allows for constant-time insertions or removals, but only sequential access of elements. Because linked lists need extra storage for references, they are impractical for lists of small data items such as characters. Unlike dynamic arrays, arbitrary number of items can be added to the linked list (limited by the memory of course) without the need to realocate, which is an expensive operation.

using System;

using System.Collections.Generic;

public class LinkedListExample

{

static void Main()

{

LinkedList<int> nums = new LinkedList<int>();

nums.AddLast(23);

nums.AddLast(34);

nums.AddLast(33);

nums.AddLast(11);

nums.AddLast(6);

nums.AddFirst(9);

nums.AddFirst(7);

LinkedListNode<int> node = nums.Find(6);

nums.AddBefore(node, 5);

foreach(int num in nums)

{

Console.WriteLine(num);

}

}

}

This is a LinkedList example with some of its methods.

LinkedList<int> nums = new LinkedList<int>();

This is an integer LinkedList.

nums.AddLast(23);

...

nums.AddFirst(7);

We populate the linked list using the AddLast() and AddFirst() methods.

LinkedListNode<int> node = nums.Find(6);

nums.AddBefore(node, 5);

A LinkedList consists of nodes. We find a specific node and add an element before it.

foreach(int num in nums)

{

Console.WriteLine(num);

}

We are printing all elements to the console.

**Dictionary**

A dictionary, also called an associative array, is a collection of unique keys and a collection of values, where each key is associated with one value. Retrieving and adding values is very fast. Dictionaries take more memory, because for each value there is also a key.

using System;

using System.Collections.Generic;

public class DictionaryExample

{

static void Main()

{

Dictionary<string, string> domains = new Dictionary<string, string>();

domains.Add("de", "Germany");

domains.Add("sk", "Slovakia");

domains.Add("us", "United States");

domains.Add("ru", "Russia");

domains.Add("hu", "Hungary");

domains.Add("pl", "Poland");

Console.WriteLine(domains["sk"]);

Console.WriteLine(domains["de"]);

Console.WriteLine("Dictionary has {0} items",

domains.Count);

Console.WriteLine("Keys of the dictionary:");

List<string> keys = new List<string>(domains.Keys);

foreach(string key in keys)

{

Console.WriteLine("{0}", key);

}

Console.WriteLine("Values of the dictionary:");

List<string> vals = new List<string>(domains.Values);

foreach(string val in vals)

{

Console.WriteLine("{0}", val);

}

Console.WriteLine("Keys and values of the dictionary:");

foreach(KeyValuePair<string, string> kvp in domains)

{

Console.WriteLine("Key = {0}, Value = {1}",

kvp.Key, kvp.Value);

}

}

}

We have a dictionary where we map domain names to their country names.

Dictionary<string, string> domains = new Dictionary<string, string>();

We create a dictionary with string keys and values.

domains.Add("de", "Germany");

domains.Add("sk", "Slovakia");

domains.Add("us", "United States");

...

We add some data to the dictionary. The first string is the key. The second is the value.

Console.WriteLine(domains["sk"]);

Console.WriteLine(domains["de"]);

Here we retrieve two values by their keys.

Console.WriteLine("Dictionary has {0} items",

domains.Count);

We print the number of items by referring to the Count property.

List<string> keys = new List<string>(domains.Keys);

foreach(string key in keys)

{

Console.WriteLine("{0}", key);

}

These lines retrieve all keys from the dictionary.

List<string> vals = new List<string>(domains.Values);

foreach(string val in vals)

{

Console.WriteLine("{0}", val);

}

These lines retrieve all values from the dictionary.

foreach(KeyValuePair<string, string> kvp in domains)

{

Console.WriteLine("Key = {0}, Value = {1}",

kvp.Key, kvp.Value);

}

Finally, we print both keys and values of the dictionary.

$ ./dictionary.exe

Slovakia

Germany

Dictionary has 6 items

Keys of the dictionary:

de

sk

us

ru

hu

pl

Values of the dictionary:

Germany

Slovakia

United States

Russia

Hungary

Poland

Keys and values of the dictionary:

Key = de, Value = Germany

Key = sk, Value = Slovakia

Key = us, Value = United States

Key = ru, Value = Russia

Key = hu, Value = Hungary

Key = pl, Value = Poland

This is the output of the example.

**Queues**

A queue is a First-In-First-Out (FIFO) data structure. The first element added to the queue will be the first one to be removed. Queues may be used to process messages as they appear or serve customers as they come. The first customer which comes should be served first.

using System;

using System.Collections.Generic;

public class QueueExample

{

static void Main()

{

Queue<string> msgs = new Queue<string>();

msgs.Enqueue("Message 1");

msgs.Enqueue("Message 2");

msgs.Enqueue("Message 3");

msgs.Enqueue("Message 4");

msgs.Enqueue("Message 5");

Console.WriteLine(msgs.Dequeue());

Console.WriteLine(msgs.Peek());

Console.WriteLine(msgs.Peek());

Console.WriteLine();

foreach(string msg in msgs)

{

Console.WriteLine(msg);

}

}

}

In our example, we have a queue with messages.

Queue<string> msgs = new Queue<string>();

A queue of strings is created.

msgs.Enqueue("Message 1");

msgs.Enqueue("Message 2");

...

The Enqueue() adds a message to the end of the queue.

Console.WriteLine(msgs.Dequeue());

The Dequeue() method removes and returns the item at the beginning of the queue.

Console.WriteLine(msgs.Peek());

The Peek() method returns the next item from the queue, but does not remove it from the collection.

$ ./queue.exe

Message 1

Message 2

Message 2

Message 2

Message 3

Message 4

Message 5

The Dequeue() method removes the "Message 1" from the collection. The Peek() method does not. The "Message 2" remains in the collection.

**Stacks**

A *stack* is a Last-In-First-Out (LIFO) data structure. The last element added to the queue will be the first one to be removed. The C language uses a stack to store local data in a function. The stack is also used when implementing calculators.

using System;

using System.Collections.Generic;

public class StackExample

{

static void Main()

{

Stack<int> stc = new Stack<int>();

stc.Push(1);

stc.Push(4);

stc.Push(3);

stc.Push(6);

stc.Push(4);

Console.WriteLine(stc.Pop());

Console.WriteLine(stc.Peek());

Console.WriteLine(stc.Peek());

Console.WriteLine();

foreach(int item in stc)

{

Console.WriteLine(item);

}

}

}

We have a simple stack example above.

Stack<int> stc = new Stack<int>();

A Stack data structure is created.

stc.Push(1);

stc.Push(4);

...

The Push() method adds an item at the top of the stack.

Console.WriteLine(stc.Pop());

The Pop() method removes and returns the item from the top of the stack.

Console.WriteLine(stc.Peek());

The Peek() method returns the item from the top of the stack. It does not remove it.

$ ./stack.exe

4

6

6

6

3

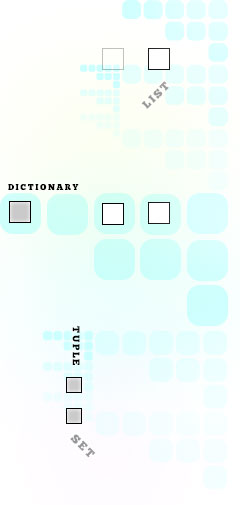
4

1

This is the output of the stack.exe program.

**C# Collections**

[**Array**](http://www.dotnetperls.com/array) [**Class**](http://www.dotnetperls.com/class) **Collections** [**File**](http://www.dotnetperls.com/file) [**String**](http://www.dotnetperls.com/string) [**.NET**](http://www.dotnetperls.com/net) [Algorithm](http://www.dotnetperls.com/algorithm) [ASP.NET](http://www.dotnetperls.com/asp)[Cast](http://www.dotnetperls.com/cast) [Compression](http://www.dotnetperls.com/compression) [Data](http://www.dotnetperls.com/data) [Delegate](http://www.dotnetperls.com/delegate) [Directive](http://www.dotnetperls.com/directive) [Enum](http://www.dotnetperls.com/enum) [Exception](http://www.dotnetperls.com/exception) [If](http://www.dotnetperls.com/if)[Interface](http://www.dotnetperls.com/interface) [Keyword](http://www.dotnetperls.com/keyword) [LINQ](http://www.dotnetperls.com/linq) [Loop](http://www.dotnetperls.com/loop) [Method](http://www.dotnetperls.com/method) [Number](http://www.dotnetperls.com/number) [Regex](http://www.dotnetperls.com/regex) [Sort](http://www.dotnetperls.com/sort)[StringBuilder](http://www.dotnetperls.com/stringbuilder) [Struct](http://www.dotnetperls.com/struct) [Switch](http://www.dotnetperls.com/switch) [Time](http://www.dotnetperls.com/time) [Value](http://www.dotnetperls.com/valuetype) [Windows](http://www.dotnetperls.com/windows) [WPF](http://www.dotnetperls.com/wpf)

**[](http://www.dotnetperls.com/)**

At its simplest, an object holds a single value. At its most complex, it holds references to many other objects. The .NET Framework provides **collections**: these include List and Dictionary. They are often useful.

Data types comprising collections of abstract objects are a central object of study in computer science.**Sedgewick, p. 143**

**List**

First we look at the List. This type provides an efficient and dynamically-allocated array. It does not provide fast lookup in the general case (the Dictionary is better for lookups). List is excellent when used in loops.

[**List**](http://www.dotnetperls.com/list)

**Program that uses List type: C#**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Use the List type.

**List**<string> list = new List<string>();

list.Add("cat");

list.Add("dog");

foreach (string element in list)

{

Console.WriteLine(element);

}

}

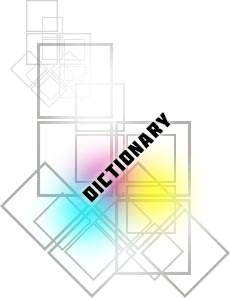
}

**Output**

cat

dog

**Dictionary**



The Dictionary type is important. It is an implementation of a hash table: an extremely efficient way to store keys for lookup. Dictionary is fast. It is well-designed and reliable. It works well in many programs.

[**Dictionary**](http://www.dotnetperls.com/dictionary)

**Note:**This program shows how the Dictionary is used with type parameters to store keys and values of specific types.

**Tip:**Many of the most powerful (and fastest) collections are found in System.Collections.Generic: these are generic types.

[**Generic Class**](http://www.dotnetperls.com/generic)

**Program that uses Dictionary: C#**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Use the dictionary.

**Dictionary**<string, int> dict = new Dictionary<string, int>();

dict.Add("cat", 1);

dict.Add("dog", 4);

Console.WriteLine(dict["cat"]);

Console.WriteLine(dict["dog"]);

}

}

**Output**

1

4

**ArrayList**



Next, the ArrayList is a collection found in System.Collections. It stores objects of any type. There is no need to worry about the types of elements. But you must cast them when you need to use them. This is inconvenient.

[**ArrayList**](http://www.dotnetperls.com/arraylist)

**Program that uses System.Collections: C#**

using System;

using System.Collections;

class Program

{

static void Main()

{

**ArrayList** list = new ArrayList();

list.Add("cat");

list.Add(2);

list.Add(false);

foreach (var element in list)

{

Console.WriteLine(element);

}

}

}

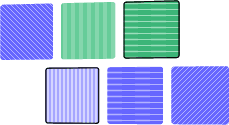
**Output**

cat

2

False

**Hashtable**



Let use continue with Hashtable. This is a lookup data structure that uses a hash code to quickly find elements. The newer Dictionary collection is usually more appropriate for programs when available.

[**Hashtable**](http://www.dotnetperls.com/hashtable)

**Caution:**I have found no reason to write a new program using Hashtable in many years. It is inferior to Dictionary.

**Program that uses Hashtable: C#**

using System;

using System.Collections;

class Program

{

static void Main()

{

**Hashtable** table = new Hashtable();

table["one"] = 1;

table["two"] = 2;

// ... Print value at key.

Console.WriteLine(table["one"]);

}

}

**Output**

1

**Lists**



Lists are linear. One element is stored after the other. The List generic type is often the best implementation available for its purpose in the .NET Framework. There are other versions of lists.

[**LinkedList**](http://www.dotnetperls.com/linkedlist)[**SortedList**](http://www.dotnetperls.com/sortedlist)

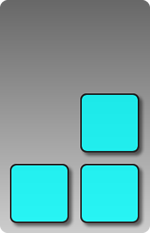
**Also:**A linear collection, such as an array or List, can be wrapped and made read-only with ReadOnlyCollection.

[**ReadOnlyCollection**](http://www.dotnetperls.com/readonlycollection)

**Performance:**Many collections (including List and Dictionary) can be optimized with a capacity. This allocates extra initial memory.

[**Capacity**](http://www.dotnetperls.com/capacity)

**Tables**



There are many versions of lookup data structures in the .NET Framework other than the Dictionary and Hashtable. Usually a Dictionary is the best option. But sometimes custom features are needed.

[**ListDictionary**](http://www.dotnetperls.com/listdictionary)[**HybridDictionary**](http://www.dotnetperls.com/hybriddictionary)[**SortedDictionary**](http://www.dotnetperls.com/sorteddictionary)[**StringDictionary**](http://www.dotnetperls.com/stringdictionary)[**NameValueCollection**](http://www.dotnetperls.com/namevaluecollection)[**DictionaryEntry**](http://www.dotnetperls.com/dictionaryentry)

**Also:**We implement a MultiMap class.  
We use the generic type features in the language.  
Our implementation is not ideal.

[**MultiMap**](http://www.dotnetperls.com/multimap)

**Performance:**Lookup tables offer many performance advantages. One overlooked benefit is that they offer fast removal of elements

# Difference between Generics and Collections with example

*Posted By : Shailendra Chauhan, 30 Dec 2012*

*Updated On : 15 Sep 2013*

*Version Support : C# 2.0, 3.0, 4.0, 5.0*

*Keywords : difference between generic collections and non-generic collections, collections vs generics, advantage of generics over collections*

Generics provides the type safe code with re-usability like as algorithm. In algorithms such as sorting, searching, comparing etc. you don’t specify what data type(s) the algorithm operates on. The algorithm can be operates with any types of data. In the same way Generics operate, you can provide different data type to Generics. For example, a sorting algorithm can operates on integer type, decimal type, string type, DateTime type etc.

In this article, I will demonstrate the advantage of Generics over Collections. Following are the main advantage of Generics.

## Code Re-usability with Generics

Suppose, you required to sort the integer and floating type numbers, then let's see how to do in collections and generics.

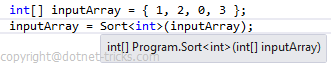
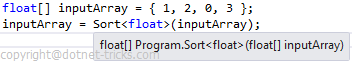
### How to do it using Collections

1. ***//Overloaded sort methods***
2. **private int[] Sort(int[] inputArray)**
3. **{**
4. ***//Sort array***
5. ***//and return sorted array***
6. **return inputArray;**
7. **}**
8. **private float[] Sort(float[] inputArray)**
9. **{**
10. ***//Sort array***
11. ***//and return sorted array***
12. **return inputArray;**
13. **}**

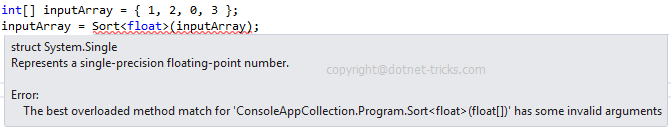
### How to do it using Generics

1. **private T[] Sort(T[] inputArray)**
2. **{**
3. ***//Sort array***
4. ***//and return sorted array***
5. **return inputArray;**
6. **}**

Here, T is short for Type and can be replaced with the Type defined in the C# language at runtime. So once we have this method, we can call it with different data types as follows and can see the beauty of Generics. In this way Generics provide code re-usability.

Now if you thinking you can make fool to the compiler by passing an integer array while it is asking for a float, you are wrong. Compiler will shows the error at compile time like as:



## Type Safety with Generics

Suppose, you want to make a list of students, then let's see how to do in collections and generics.

### How to do it using Collections

In collections we can use ArrayList to store a list of Student objects like as:

1. **class Student**
2. **{**
3. **public int RollNo{get; set;}**
4. **public string Name{get; set;}**
5. **}**
7. ***//List of students***
8. **ArrayList studentList = new ArrayList();**
9. **Student objStudent = new Student();**
10. **objStudent.Name = "Rajat";**
11. **objStudent.RollNo = 1;**
13. **studentList.Add(objStudent);**
15. **objStudent = new Student();**
16. **objStudent.Name = "Sam";**
17. **objStudent.RollNo = 2;**
19. **studentList.Add(objStudent);**
21. **foreach (Object s in studentList)**
22. **{**
23. ***//Type-casting. If s is anything other than a student***
24. **Student currentStudent = (Student)s;**
25. **Console.WriteLine("Roll # " + currentStudent.RollNo + " " + currentStudent.Name);**
26. **}**

### Problem with Collections

Suppose by mistake you have added a string value to ArrayList like as

1. **studentList.Add("Generics"); *//Fooling the compiler***

Since ArrayList is a loosely typed collection and it never ensure compile-time type checking. Hence above statement will compile without error but it will throw an InvalidCastException at run time when you try to cast it to Student Type.

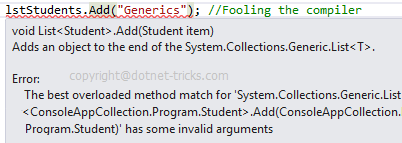
### How to do it using Generics

In generics we can use generic List to store a list of Student objects like as:

1. **List<Student> lstStudents = new List<Student>();**
3. **Student objStudent = new Student();**
4. **objStudent.Name = "Rajat";**
5. **objStudent.RollNo = 1;**
7. **lstStudents.Add(objStudent);**
9. **objStudent = new Student();**
10. **objStudent.Name = "Sam";**
11. **objStudent.RollNo = 2;**
13. **lstStudents.Add(objStudent);**
15. ***//Looping through the list of students***
16. **foreach (Student currentSt in lstStudents)**
17. **{**
18. ***//no need to type cast since compiler already knows that everything inside***
19. ***//this list is a Student***
20. **Console.WriteLine("Roll # " + currentSt.RollNo + " " + currentSt.Name);**
21. **}**

### Advantage with Generics

In case of collections you can make fool to compiler but in case of generics you can't make fool to compiler as shown below. Hence Generics provide Type Safety.



## Cleaner Code with Generics

Since compiler enforces type safety with Generics. Hence fetching data from Generics doesn't required type casting which means your code is clean and easier to write and maintain.

## Better Performance with Generics

As you have seen in above example, at the time of fetching the data from the ArrayList collection we need to do type casting which cause performance degrades. But at the time of fetching the data from the generic List we don't required to do type casting. In this way Generics provide better performance than collections.