**C# collections**

The .NET framework provides specialized classes for data storage and retrieval.  Collections are enhancement to the arrays.

There are three distinct collection types in C#:

* standard
* generic
* concurrent

The standard collections are found under the System.Collections. They do not store elements as specifically typed objects, but as objects of type Object. Standard collections include ArrayList,Hashtable, Queue, and Stack.

The generic collections are found under System.Collections.Generic. Generic collections are more flexible and are the preferred way to work with data. Generics enhance code reuse, type safety, and performance. The generic collections include Dictionary<T, T>, List<T>, Queue<T>, SortedList<T>, and Stack<T>.

Concurrent collections include BlockingCollection<T>, ConcurrentDictionary<T, T>, ConcurrentQueue<T>, and ConcurrentStack<T>.

## C# ArrayList

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.

Program.cs

using System;

using System.Collections;

namespace ArrayListEx

{

class Empty { }

class Program

{

static void Main(string[] args)

{

var data = new ArrayList();

data.Add("Visual Basic");

data.Add(344);

data.Add(55);

data.Add(new Empty());

data.Remove(55);

foreach (object el in data)

{

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.

var data = new ArrayList();

An ArrayList collection is created.

data.Add("Visual Basic");

data.Add(344);

data.Add(55);

data.Add(new Empty());

data.Remove(55);

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

data.Remove(55);

We remove one element with the Remove() method.

foreach(object el in data)

{

Console.WriteLine(el);

}

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

$ dotnet run

Visual Basic

344

ArrayListEx.Empty

This is the output of the example.

## C# List

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

Program.cs

using System;

using System.Collections.Generic;

namespace ListEx

{

class Program

{

static void Main(string[] args)

{

var 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.

var 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.

$ dotnet run

True

C#

C

False

C++

Haskell

Java

Javascript

Ruby

This is the outcome of the example.

## C# SortedList

SortedList<T, T> represents a collection of key/value pairs that are sorted.

Program.cs

using System;

using System.Collections.Generic;

namespace SortedListEx

{

class Program

{

static void Main(string[] args)

{

var sorted = new SortedList<string, int>();

sorted.Add("coins", 3);

sorted.Add("books", 41);

sorted.Add("spoons", 5);

if (sorted.ContainsKey("books"))

{

Console.WriteLine("There are books in the list");

}

foreach (var pair in sorted)

{

Console.WriteLine(pair);

}

}

}

}

The example uses a sorted list to organize items.

var sorted = new SortedList<string, int>();

The sorted list has string keys and integer values.

if (sorted.ContainsKey("books"))

{

Console.WriteLine("There are books in the list");

}

With ContainsKey() we check if there are books in the collection.

foreach (var pair in sorted)

{

Console.WriteLine(pair);

}

With foreach loop we go throug the collection and print its pairs.

$ dotnet run

There are books in the list

[books, 41]

[coins, 3]

[spoons, 5]

This is the output.

## C# 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.

Program.cs

using System;

using System.Collections.Generic;

namespace LinkedListEx

{

class Program

{

static void Main(string[] args)

{

var 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.

var 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.

$ dotnet run

7

9

23

34

33

11

5

6

This is the output.

## C# 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.

Program.cs

using System;

using System.Collections.Generic;

namespace DictionaryEx

{

class Program

{

static void Main(string[] args)

{

var 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:");

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

foreach (string key in keys)

{

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

}

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

var 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.

var 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.

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

foreach(string key in keys)

{

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

}

These lines retrieve all keys from the dictionary.

var 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.

$ dotnet run

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.

## C# 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.

Program.cs

using System;

using System.Collections.Generic;

namespace QueueEx

{

class Program

{

static void Main(string[] args)

{

var 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.

var 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.

$ dotnet run

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.

## C# 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.

Program.cs

using System;

namespace StackEx

{

class Program

{

static void Main(string[] args)

{

var myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(4);

myStack.Push(3);

myStack.Push(6);

myStack.Push(4);

Console.WriteLine(myStack.Pop());

Console.WriteLine(myStack.Peek());

Console.WriteLine(myStack.Peek());

Console.WriteLine();

foreach (int item in myStack)

{

Console.WriteLine(item);

}

}

}

}

We have a simple stack example above.

var myStack = new Stack<int>();

A Stack data structure is created.

myStack.Push(1);

myStack.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(myStack.Peek());

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

$ dotnet run

4

6

6

6

3

4

1

This is the output of the program.

This part of the C# tutorial was dedicated to Collections in C#.