# Object-oriented programming (OOP) Lecture 8: Collections, Generics, and Enum

## Collections

* C# collection types are designed to store, manage and manipulate similar data more efficiently. Data manipulation includes adding, removing, finding, and inserting data in the collection.
* Collection types implement the following common functionality:
* **Adding** and **inserting** items to a collection
* **Removing** items from a collection
* **Finding**, **sorting**, and **searching** items
* **Replacing** items
* **Copy** and **clone** collections and items
* **Capacity** and **Count** properties to find the capacity of the collection and the number of items in the collection
* .NET supports two types of collections, **generic collections**, and **non-generic collections**. Before NET 2.0, it was just collections, and when generics were added to .NET, generics collections were added as well.
* The following table lists and matches these classes.

|  |  |
| --- | --- |
| **Non-generic** | **Generic** |
| ArrayList | List |
| HashTable | Dictionary |
| SortedList | SortedList |
| Stack | Stack |
| Queue | Queue |

## 1. Non-Generic

* In non-generic collections, each element can represent a value of a **different** type. The collection **size is not fixed**. Items from the collection can be **added** or **removed** at **runtime**.

## C# ArrayList

* ArrayList class is a collection used for any types or objects.
* Array is a **fixed** size collection whereas an ArrayList's size can be **dynamically** increased or decreased.
* An Array is a collection of variables of the **same type** whereas an ArrayList is a collection of variables of the **same type or multiple types**.
* The **capacity** of an ArrayList is the number of elements the ArrayList can hold. As elements are added to an ArrayList, the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling TrimToSize or by setting the Capacity property explicitly.
* Elements in an ArrayList collection can be accessed using an **integer index**. Indexes in this collection are **zero-based**.
* The ArrayList collection **accepts null** as a valid value, and **allows duplicate elements**

## How to create an ArrayList?

Just we need to create an object form this class as shown in below example

ArrayList myList = new ArrayList();

## How to add an element to an ArrayList?

Use Add() method which will append the new element to the end of the ArrayList as shown in the below example

myList.Add(“Mukalla”);

myList.Add(2024);

## How to read element from an ArrayList?

By using foreach loop we can read the elements of an ArrayList, **OR** we can use the index of the required element as shown in the below example

using System.Collections;

class Program

{

static void Main(string[] args)

{

ArrayList myList=new ArrayList();

myList.Add(101);

myList.Add("Hani");

string city = "Mukalla";

myList.Add(city);

Console.WriteLine("element at first index is: "+myList[0]);

Console.WriteLine("all elements in myList are : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

}

}

## How to insert an element to a C# ArrayList

Add() method will append the element at the end of the ArrayList, but if we want to insert the new element at a specific location, then we can use insert() method and give it the new element to be inserted and the index in which it should be inserted as shown in the following example to insert the value 2024 at **third** position (i.e. [2] index )

using System.Collections;

class Program

{

static void Main(string[] args)

{

ArrayList myList=new ArrayList();

myList.Add(101);

myList.Add("Hani");

myList.Insert(2,2024);

string city = "Mukalla";

myList.Add(city);

Console.WriteLine("all elements in myList are : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

}

}

## How to remove an element from an ArrayList

We can remove single element or group of elements form ArrayList by using one of the following four methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| Remove | To remove single element from the ArrayList we just need to pass that element to this method |
| RemoveAt | To remove single element from an ArrayList by specifying its index in this methods. |
| RemoveRange | To remove group of elements at specific index range. **It accepts two parameters**. The first parameter is the index from which the remove should starts and the second parameter is the count of elements to be removed. |
| Clear | To remove all elements from the ArrayList, **but** doesn’t reduce the capacity. |

using System.Collections;

class Program

{

static void Main(string[] args)

{

ArrayList myList=new ArrayList();

myList.Add(101);

myList.Add("Hani");

myList.Insert(2,2024);

string city = "Mukalla";

myList.Add(city);

Console.WriteLine("myList before remove : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

myList.RemoveRange(1,2);

Console.WriteLine("myList after remove : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

}

}

## How to sort an ArrayList

We can use Sort() method to sort the elements of an ArrayList in an ascending order

using System.Collections;

class Program

{

static void Main(string[] args)

{

ArrayList myList=new ArrayList();

myList.Add("zaki");

myList.Add("Hani");

myList.Add("Murad");

myList.Add("Ahmed");

Console.WriteLine("myList before sort : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

myList.Sort();

//myList.Reverse();

Console.WriteLine("myList after sort : ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

}

}

**Note**: Arraylist allocates memory for **4** items whenever an object is created. When a fifth item is added, memory for another 4 items are added.

**Capacity**: is a property that returns the number of items for which memory is allocated. Whereas **count** returns the number of items added in the list.

using System.Collections;

class Program

{

static void Main(string[] args)

{

ArrayList myList=new ArrayList();

Console.WriteLine("capacity of my list before adding any element: " + myList.Capacity);

Console.WriteLine("number of elements in my list before adding any element: " + myList.Count);

myList.Add(001);

Console.WriteLine("capacity after adding first element is: " + myList.Capacity);

Console.WriteLine("number of elements in my list after adding first element is: " + myList.Count);

myList.Add(002);

myList.Add(003);

myList.Add(004);

Console.WriteLine("capacity after adding 4 elements is: " + myList.Capacity);

Console.WriteLine("number of elements in my list after adding 4 element is: " + myList.Count);

myList.Add(005);

Console.WriteLine("capacity after adding 5th element is: " + myList.Capacity);

Console.WriteLine("number of elements in my list after adding 5th element is: " + myList.Count);

Console.WriteLine("-----all elements in myList ---- ");

foreach (object element in myList)

{

Console.WriteLine(element);

}

}

}

## C# Hashtable

* Hashtable is similar to arraylist but represents the items as a combination of a **key and value** pairs.

## Declaring a Hashtable

By creating an object from this class as shown below

Hashtable ht = new Hashtable();

## Methods of Hashtable

Some of the important methods of a hashtable are shown in the table below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| Add | Adds an element with the specified key and value in the Hashtable. |
| Remove | Remove an element based on its key |
| Clear | Removes all the elements in the Hashtable |
| ContainsKey | Determined whether the Hashtable contains a specified key or not. |
| ContainsValue | Determined whether the Hashtable contains a specified value or not. |

using System.Collections;

class Program

{

static void Main(string[] args)

{

Hashtable Employee = new Hashtable();

Employee.Add("ID",101);

Employee.Add("Name", "Hani");

Employee.Add("salary", 100000);

Employee.Add("job", "Developer");

//we can loop over keys only

Console.WriteLine("--- keys only ---");

foreach(object k in Employee.Keys)

{

Console.WriteLine(k);

}

//we can loop over values only

Console.WriteLine("--- values only ---");

foreach (object v in Employee.Values)

{

Console.WriteLine(v);

}

//we can loop over both keys and values using DictionaryEntry

Console.WriteLine("--- both keys and values ---");

foreach (DictionaryEntry emp in Employee)

{

Console.WriteLine(emp.Key +"-->"+emp.Value);

}

}

}

**Note:**

A hash table does not maintain an ordered collection; there is no specific order to the collection of keys or values obtained. Each element is a key/value pair stored in a **DictionaryEntry** object. **A key cannot be a null, but a value can be**.

## C# SortedList

1. It is a class that has the combination of arraylist and hashtable.
2. Represents the data as a key and value pair.
3. Arranges all the items in sorted order.

using System.Collections;

class Program

{

static void Main(string[] args)

{

SortedList employee= new SortedList();

employee.Add("Id", 101);

employee.Add("Name", "Hani");

employee.Add("job", "Developer");

employee.Add("salary", 100000);

foreach(DictionaryEntry emp in employee)

{

Console.WriteLine(emp.Key+"-->"+emp.Value);

}

}

}

## SortedList Methods

|  |  |
| --- | --- |
| Add(Object, Object) | Adds an element with the specified key and value to a SortedList object. |
| Clone() | Creates a shallow copy of a SortedList object. |
| Contains(Object) | Return bool value, Determines whether the SortedList contains a specific key. |
| ContainsKey(Object) | Return bool value, Determines whether the SortedList contains a specific key. |
| ContainsValue(Object) | Return bool value, Determines whether the SortedList contains a specific value. |
| IndexOfKey(Object) | Returns the zero-based index of the specified key in a SortedList object. |
| IndexOfValue(Object) | Returns the zero-based index of the first occurrence of the specified value in a SortedListobject. |

## C# Stack

A stack is a data structure in which items are added or removed in a **Last In First Out** (**LIFO**) manner. That means that items that are added last will be removed first. We can use an example of stack of plates in which the last added plate will be removed first.

The basic operations that can be performed on a stack are:

1. **Push**: Push in a stack means adding elements to the stack. Elements are added in the stack from one direction only. I am calling that position as the top of the stack
2. **Pop**: Pop in a stack means removing an element from the stack. Items are removed from the stack from one direction, in other words from the top. In other words, if items are added in the order 1, 2, 3 then on calling pop then item 3 will be removed first then 2 and then 1.
3. **Peek**: A Peek operation returns the last added element in the stack or the top element of the stack.
4. **Clear**: It will clear all the items of the stack.

using System.Collections;

class Program

{

static void Main(string[] args)

{

Stack stk = new Stack();

stk.Push("cs.net");

stk.Push("vb.net");

stk.Push("asp.net");

stk.Push("sqlserver");

Console.WriteLine("count of elements in my list is: " + stk.Count);

foreach (object obj in stk)

{

Console.WriteLine(obj);

}

Console.WriteLine("peek element is: " + stk.Peek());

Console.WriteLine("removed item is : "+stk.Pop());

stk.Pop();

foreach (object obj in stk)

{

Console.WriteLine(obj);

}

}

}

## C# Queue

A Queue in C# represents a **first-in, first-out** (**FIFO**) collection of objects. An example of a queue is a *line of people waiting* for something.

## How to add elements to a queue?

By using Enqueue() method we can adds an object to the end of the Queue

## How to remove an item from a queue?

## By using Dequeue() method which Removes and returns the object at the beginning of the Queue

using System.Collections;

class Program

{

static void Main(string[] args)

{

Queue q = new Queue();

q.Enqueue("cs.net");

q.Enqueue("vb.net");

q.Enqueue("asp.net");

q.Enqueue("sqlserver");

Console.WriteLine("count of elements in my queue is: " + q.Count);

foreach (object obj in q)

{

Console.WriteLine(obj);

}

Console.WriteLine("The remove element from queue is "+q.Dequeue());

}

}

## 2. Generic Collections

* Generic Collections work on the specific type specified in the program, whereas non-generic collections work on the object type.
* Specific type
* Array Size is not fixed
* Elements can be added/removed at runtime.

Before understanding generic collections, let’s start our journey by understanding the generic concept first.

Suppose you want to write a method that adds two integer numbers and return the result. You can do so as shown in the following code

using System.Collections;

class Program

{

public int Add(int num1, int num2)

{

return num1 + num2;

}

static void Main(string[] args)

{

Program program = new Program();

Console.WriteLine("The Result is: "+program.Add(10,20));

}

}

Now suppose you need to add two float numbers instead of integer numbers. In this case we can’t use the same method declared above to do this job, so we have to create a new method that takes two parameters of type float as shown below

using System.Collections;

class Program

{

public int Add(int num1, int num2)

{

return num1 + num2;

}

public float Add(float num1, float num2)

{

return num1 + num2;

}

static void Main(string[] args)

{

Program program = new Program();

Console.WriteLine("The Result of integer numbers is: "+program.Add(10,20));

Console.WriteLine("The Result of float numbers is: " + program.Add(10.5f, 20.3f));

}

}

Likewise, we have to create as many methods as different data types which will increase the length of the code and there is no reusability for the code.

**To solve the above problem, a very beautiful concept called generic was introduced in C# 2**

Using this concept we can create a method or a class with general type. At the time of creating an object of the class or calling the method we need to specify the required type.

## Generic Methods in C#

The example below shows how the above problem is solved

using System.Collections;

class Program

{

public T Add<T>(T num1, T num2)

{

dynamic a = num1;

dynamic b = num2;

return a + b;

}

static void Main(string[] args)

{

Program program = new Program();

Console.WriteLine("The Result of integer numbers is: "+program.Add(10,20));

Console.WriteLine("The Result of float numbers is: " + program.Add(10.5f, 20.3f));

}

}

**NOTE**: as you can see from the above example we have replaced all the specified types like int with general type T which can be any. And we can simply call the same method with different types. And there is no need to create multiple methods for different data types.

## Generic class in C#

We can use the concept of generic with classes. So instead of specifying the type for every method we can simply create a class as a generic class and then at the time of instantiation we need to specify the type for every object as shown below

using System.Collections;

class Program<T>

{

public T Add(T num1, T num2)

{

dynamic a = num1;

dynamic b = num2;

return a + b;

}

}

public class myClass {

static void Main(string[] args)

{

Program<int> program1 = new Program<int>();

Console.WriteLine("The Result of integer numbers is: " + program1.Add(10, 20));

Program<float> program2 = new Program<float>();

Console.WriteLine("The Result of float numbers is: " + program2.Add(10.5f, 20.3f));

}

}

**NOTE**

C# allows you to define generic classes, interfaces, abstract classes, fields, methods, static methods, properties, events, delegates, and operators using the [type parameter](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters) and without the specific data type. A type parameter is a placeholder for a particular type specified when creating an instance of the generic type.

Now after understanding the concept of generic, you can easily understand the generic collections.

## C# List

class Program

{

public static void Main()

{

List<int> lst = new List<int>();

lst.Add(100);

lst.Add(200);

lst.Add(300);

lst.Add(400);

Console.WriteLine("count of elements in my list is: " + lst.Count);

Console.WriteLine("count of elements in my list is: " + lst.Capacity);

foreach (int i in lst)

{

Console.WriteLine(i);

}

}

}

## C# Dictionary

class Program

{

public static void Main()

{

Dictionary<int, string> dct = new Dictionary<int, string>();

dct.Add(1, "cs.net");

dct.Add(2, "vb.net");

dct.Add(3, "vb.net");

dct.Add(4, "vb.net");

foreach (KeyValuePair<int, string> kvp in dct)

{

Console.WriteLine(kvp.Key + " " + kvp.Value);

}

}

}

## C# SortedList

class Program

{

public static void Main()

{

SortedList<string, string> sl = new SortedList<string, string>();

sl.Add("ora", "oracle");

sl.Add("vb", "vb.net");

sl.Add("cs", "cs.net");

sl.Add("asp", "asp.net");

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

{

Console.WriteLine(kvp.Key + " " + kvp.Value);

}

}

}

## C# Stack

class Program

{

public static void Main()

{

Stack<string> stk = new Stack<string>();

stk.Push("cs.net");

stk.Push("vb.net");

stk.Push("asp.net");

stk.Push("sqlserver");

foreach (string s in stk)

{

Console.WriteLine(s);

}

}

}

## C# Queue

class Program

{

public static void Main()

{

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

q.Enqueue("cs.net");

q.Enqueue("vb.net");

q.Enqueue("asp.net");

q.Enqueue("sqlserver");

foreach (string s in q)

{

Console.WriteLine(s);

}

}

}

## Enum in C#

* Enums are powerful data types in C# that allow you to define a **set of named constants**. They are often used to represent a fixed number of possible values that a variable can take on.

Here's a step-by-step tutorial on how to use enums in C#:

**Step 1**. Define an enum

* To define an enum, use the **enum** keyword followed by the name of the enum. Here's an example:

enum DaysOfWeek

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

* In this example, we've defined an enum called **DaysOfWeek** with seven possible values: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday. **Note** that the values of an enum are constants and are not allowed to change during program execution.

**Step 2.** Declare a variable of the enum type

* To declare a variable of the enum type, simply use the enum name followed by the variable name. Here's an example:

DaysOfWeek today = DaysOfWeek.Monday;

* In this example, we've declared a variable called today of type DaysOfWeek and assigned it the value of Monday.

**Step 3**. Use the enum variable

* You can use the enum variable just like any other variable. Here are some examples:

if (today == DaysOfWeek.Saturday || today == DaysOfWeek.Sunday)

{

Console.WriteLine("It's the weekend!");

}

else

{

Console.WriteLine("It's a weekday.");

}

switch (today)

{

case DaysOfWeek.Monday:

Console.WriteLine("Back to work!");

break;

case DaysOfWeek.Friday:

Console.WriteLine("Happy Friday!");

break;

default:

Console.WriteLine("Just another day...");

break;

}

* In the first example, we used if statement to check whether today is a weekend day or a weekday. In the second example, we've used a switch statement to print a different message depending on which day of the week today is.