Collections

Windows Programming Course

Agenda

1. Collection Overview
2. List<T> Type
3. Other Collection Types

1.
Collection Overview

Collection Interfaces and Types

Most collection classes are in the `System.Collections` and `System.Collections.Generic` namespaces.

- `System.Collections.Generic`: Generic collection classes
- `System.Collections.Specialized`: Collection classes that are specialized for a specifc type.

Important Interfaces

INTERFACE	DESCRIPTION
IEnumerable<T>	The interface IEnumerable is required by the `foreach` statement. This interface defnes the method `GetEnumerator`, which returns an enumerator that implements the IEnumerator interface.
ICollection<T>	ICollection<T>is implemented by generic collection classes. The interface supports: Properties: Count Methods: CopyTo, Add, Remove, Clear.
IList<T>	The IList<T> interface is for lists where elements can be accessed from their position. This interface defnes an indexer, as well as ways to insert or remove items from specifc positions (Insert, RemoveAt methods). IList<T> derives from ICollection<T>.

Important Interfaces (cont.)

INTERFACE	DESCRIPTION
IDictionary<TKey, TValue>	The interface IDictionary<TKey, TValue> is implemented by generic collection classes that have a key and a value. With this interface all the keys and values can be accessed, items can be accessed with an indexer of type key, and items can be added or removed.
ILookup<TKey, TValue>	Like the IDictionary<TKey, TValue> interface, lookups have keys and values. However, with lookups the collection can contain multiple values with one key.
IComparer<T>	The interface IComparer<T> is implemented by a comparer and used to sort elements inside a collection with the Compare method.

2.
List<T> Type

List<T> type

```
public class List<T> : IList<T>, Icollection<T>,
IEnumerable<T>, IList, ICollection, IEnumerable
```

List is a strongly typed list of objects that can be accessed by indexed.

Creating List

```
var intList = new List<int>();
var students = new List<Student>(10);
intList.Capacity = 10; // Get or set the capacity of a collection
intList.Count; // Get number of elements
intList.TrimExcess(); // get rid of the unneeded capacity
```

Initially, the capacity of the list is 0 (Empty list)

Add a new element => The capacity = 4

Add the fifth element => The capacity = 8

If 8 elements is not enough => The capacity = 16

With every resize the capacity of the list is doubled.

Collection Initializers

Assigning values to collections using collection initializers:

```
var intList = new List<int>() {1, 2};
var stringList = new List<string>() { "one", "two" };
```

Operations of List Type

- Add
- Insert
- Access
- Remove
- Search
- Sort

Operations of List Type – Adding Elements

```
var intList = new List<int>();
intList.Add(1);
intList.Add(2);

var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

Operations of List Type – Inserting Elements

```
var index = 3;
students.Insert(index, new Student(”Alice”, 3498));
```

If the index set is larger than the number of elements in the collection, an exception of type `ArgumentOutOfRangeException` is thrown.

Operations of List Type – Accessing Elements

```
student  = students[1];
for (int i = 0; i < students.Count; i++) {
  Console.WriteLine(students[i]);
}

foreach (var s in students) {
  Console.WriteLine(s);
}
```

Operations of List Type – Removing Elements

```
students.RemoveAt(3);

if (!students.Remove(alice)) {
  Console.WriteLine("object not found in collection");
}

int index = 3;
int count = 5;
students.RemoveRange(index, count);
```

Operations of List Type – Searching Elements

```
public T Find(Predicate<T> match);
public T FindLast(Predicate<T> match);
public List<T> FindAll(Predicate<T> match);
public int FindIndex(Predicate<T> match);
public int FindLastIndex(Predicate<T> match);
```

e.g.:

```
var alice = students.Find(x => x.Name.Contain("Alice"));
var studentsInCurrentYear = students.FindAll(x => x.StudentId > 1002);
```

Operations of List Type – Sorting Elements

The default Sort method without arguments is used if the elements in the collection implement the interface `IComparable`. Sort uses quick sort algorithm.

```
List<T>.Sort();
```

Other overloads of the Sort method:

```
List<T>.Sort(Comparison<T>);
List<T>.Sort(IComparer<T>);
List<T>.Sort(Int32, Int32, IComparer<T>);
```

Hands-on Exercise

Create a list of student and sort by Student ID:

- Create Student class with two properties: Name, StudentID
  - The `Student` class derives from the interface `IComparable`, implement the method to compare student by StudentID then Name
- Create a list of students, add some data and sort the list
  - Print the list to the console.

Hands-on Exercise

Implement a comparer:

- Using the exercise above
- Implement a comparer (derive from IComparer Interface) which compare first name only.
- Invoke the Sort method with new comparer.

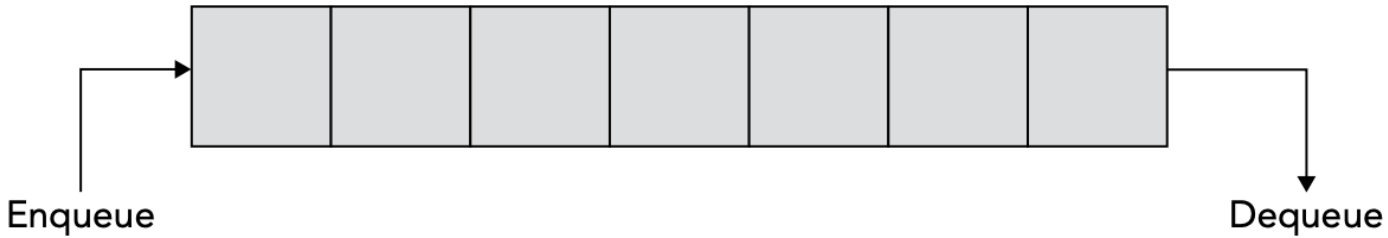
3.
Other Collection Types

Other Collection Types

- Queues
- Stacks
- Linked Lists
- Sorted List
- Dictionaries

Queues

A queue is a collection whose elements are processed *first in, first out* (FIFO), meaning the item that is put first in the queue is read first.



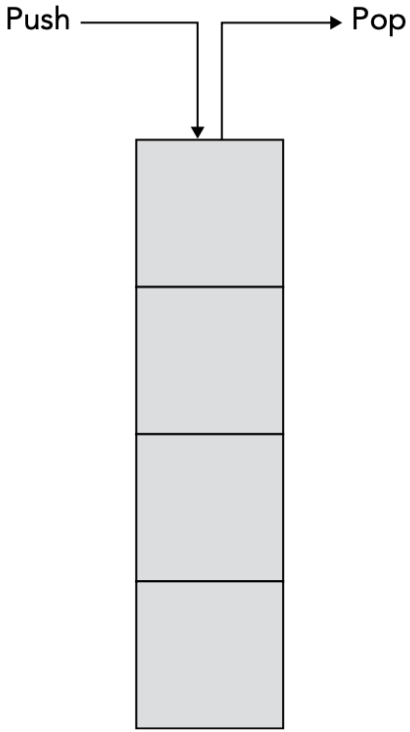
Queues (cont.)

CONSTRAINT	DESCRIPTION
`Count`	Returns the number of items in the queue.
`Enqueue`	Adds an item to the end of the queue.
`Dequeue`	Reads and removes an item from the head of the queue. If there are no more items in the queue when the Dequeue method is invoked, an exception of type `InvalidOperationException` is thrown.
`TryDequeue`	Be the same with `Dequeue` but do not throw exception if there are no more items
`Peek`	Reads an item from the head of the queue but does not remove the item.

Stacks

A stack is another container that is very similar to the queue.

Using different methods to access the stack. The item that is added last to the stack is read first, so the stack is a *last in, first out* (LIFO) container.



Stacks (cont.)

CONSTRAINT	DESCRIPTION
`Count`	Returns the number of items in the stack.
`Push`	Adds an item on top of the stack.
`Pop`	Removes and returns an item from the top of the stack. If the stack is empty, an exception of type `InvalidOperationException` is thrown.
`Peek`	Returns an item from the top of the stack but does not remove the item.
`Contains`	Checks whether an item is in the stack and returns true if it is.

Stacks (cont.)

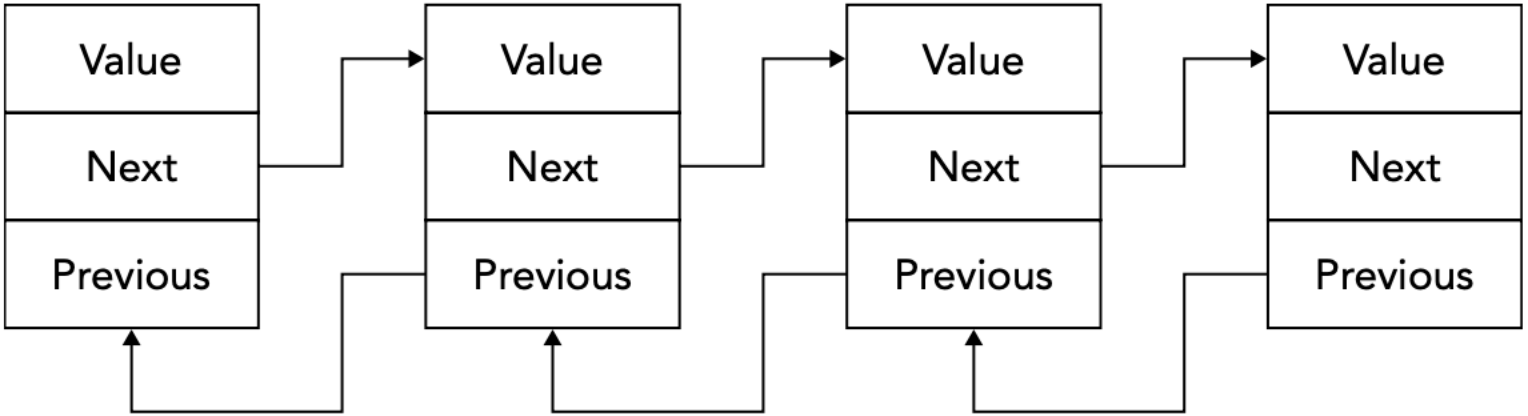
e.g.:

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

foreach (char item in alphabet) {
  Console.Write(item);
}
// Output: ???
```

Linked Lists

`LinkedList<T>` is a doubly linked list, whereby one element references the next and the previous one.



Dictionaries

The Dictionary<TKey, TValue> is a generic collection that stores key-value pairs in no particular order.

- Keys must be unique and cannot be null.
- Values can be null or duplicate.
- Values can be accessed by passing associated key in the indexer e.g. myDictionary[key]

Dictionaries – Creating

```
IDictionary<int, string> numberNames = new Dictionary<int,
string>();
numberNames.Add(1,"One");
numberNames.Add(2,"Two");
numberNames.Add(3,"Three");

var cities = new Dictionary<string, string>(){
  {"UK", "London, Manchester, Birmingham"},
  {"USA", "Chicago, New York, Washington"},
  {"India", "Mumbai, New Delhi, Pune"}
};
```

Dictionaries – Accessing

```
Console.WriteLine(cities["UK"]); //prints value of UK key
Console.WriteLine(cities["USA"]);//prints value of USA key

Console.WriteLine(cities["France"]); // run-time exception: Key does
not exist

//use ContainsKey() to check for an unknown key
if(cities.ContainsKey("France")){
    Console.WriteLine(cities["France"]);
}
```

Dictionaries – Accessing (cont.)

```
//use TryGetValue() to get a value of unknown key
string result;
if(cities.TryGetValue("France", out result)) {
  Console.WriteLine(result);
}

//use ElementAt() to retrieve key-value pair using index
for (int i = 0; i < cities.Count; i++) {
    Console.WriteLine("Key: {0}, Value: {1}",
                                  cities.ElementAt(i).Key,
                                  cities.ElementAt(i).Value);
}
```

Dictionaries – Updating

```
cities["UK"] = "Liverpool, Bristol"; // update value of UK key
cities["USA"] = "Los Angeles, Boston"; // update value of USA key
// cities["France"] = "Paris"; //throws run-time exception:
KeyNotFoundException
```

Dictionaries – Removing

```
cities.Remove("UK"); // removes UK
//cities.Remove("France"); //throws run-time exception:
KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it
    cities.Remove("France");
}

cities.Clear(); //removes all elements
```

Performance

COLLECTION	ADD	INSERT	REMOVE	ITEM	SORT	FIND
`List<T>`	O(1) or O(n) if the collection must be resized	O(n)	O(n)	O(1)	O (n log n), worst case O(n ^ 2)	O(n)
`Stack<T>`	Push, O(1), or O(n) if the stack must be resized	n/a	Pop, O(1)	n/a	n/a	n/a
`Queue<T>`	Enqueue, O(1), or O(n) if the queue must be resized	n/a	Dequeue, O(1)	n/a	n/a	n/a
`LinkList<T>`	AddLast O(1)	Add After O(1)	O(1)	n/a	n/a	O(n)
`Dictionary <TKey, TValue>`	O(1) or O(n)	n/a	O(1)	O(1)	n/a	n/a

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com