Collection

Collection

- The need of more flexible data structure (ex: dynamically growing and shrinking) was the motive for creation collection
- collections are classes that provide a convenient way to work with groups of objects

Collection

- C# collections typically implement certain key interfaces which define their behavior:
  - **IEnumerable**: Provides the ability to **iterate** through the collection.
    - Readonly Secnario
  - **ICollection**: Defines size, enumerators, and adding and removing methods for all collections.
    - Manipulation Secnario
  - **IList**: Represents a collection of objects that can be individually accessed by index (inserting , removing).
    - Advanced List Operation
  - **IDictionary<TKey, TValue>**: Represents a collection of key-value pairs.

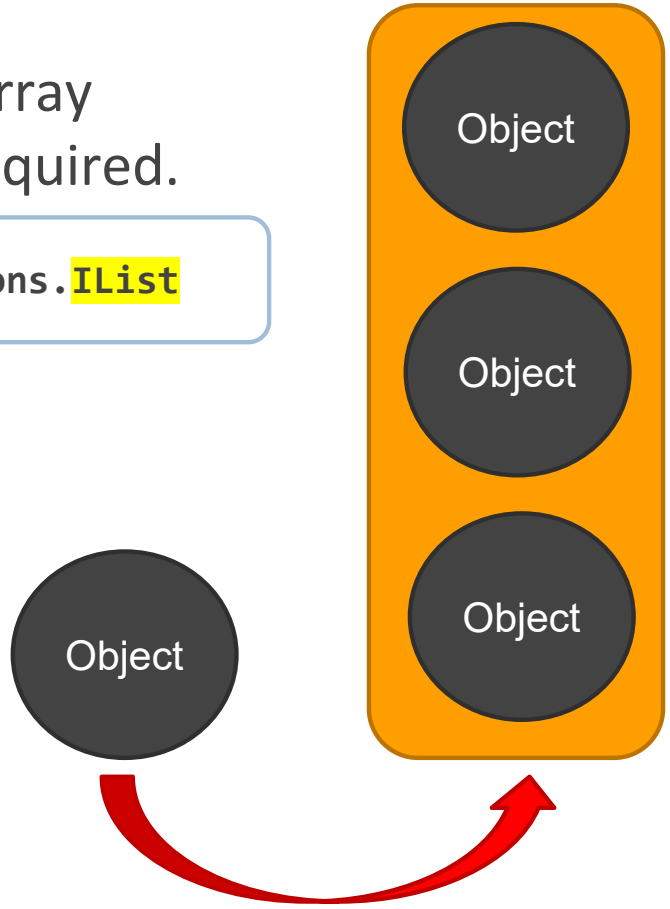
ArrayList

- Implements the IList interface using an array whose size is dynamically increased as required.

```
public class ArrayList : ICloneable, System.Collections.IList
```

- Methods
  - Add(Object)
  - Insert(Index,Object)
  - Remove(Object)
  - RemoveAt(index)
  - RemoveRange(start index, end index)
  - Clear()

```
ArrayList arlist = new ArrayList();
arlist.Add(10);
```



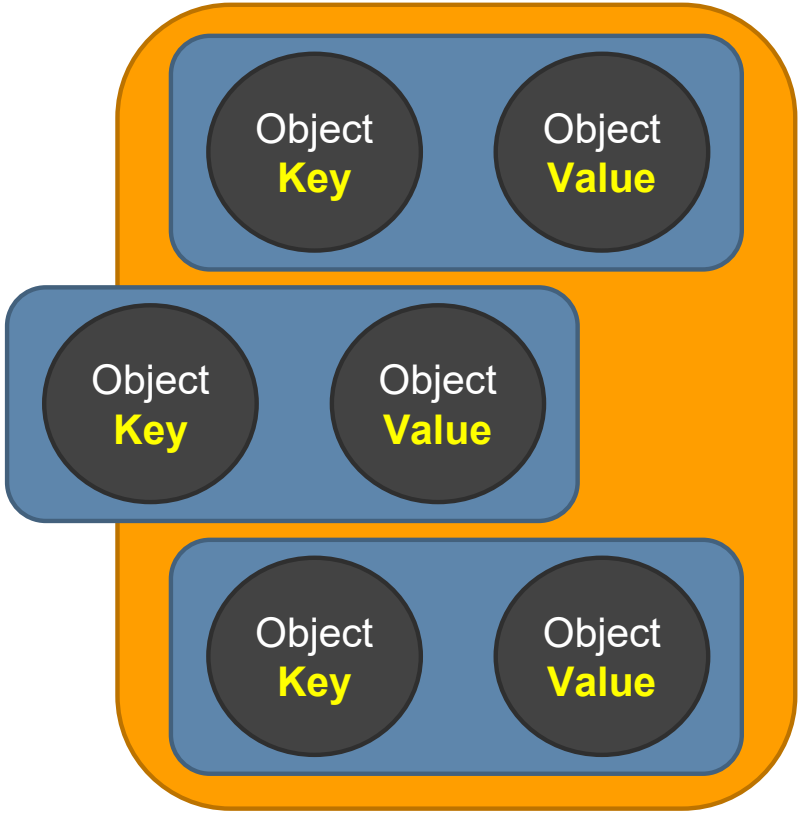
ArrayList

- Methods
  - TrimToSize( )
  - Sort()
  - Reverse()
  - Object[] ToArray()
  - int indexOf(Object)
  - Contains(Object) ➔ Object.Equals()
  - [int index] indexer
- Properties
  - Capacity
  - Count

SortedList

```
public class SortedList : ICloneable, System.Collections.IDictionary
```

- Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.
- Collection of (key-value) pair where key Is unique
- Internally maintains two arrays
  - One for keys one for values
- Auto sorted by
  - key's implementation of *IComparable*
  - using *IComparer* Implementation
    - Passed to Constructor



SortedList

- Mehods
  - Add(Object Key, Object item)
  - Clear()
  - ContainsValue()
  - ContainsKey()
  - IndexOfKey(key)
  - indexOfValue(Value)
  - RemoveAt(index)
  - TryGetValue(Tkey,out Value)
  - GetByIndex(index)
  - GetKey(index)
- Access to elements through *key* or through *index*

```
SortedList sl=new SortedList();
sl.Add(1,1);
sl.Add(5,5);
System.Console.WriteLine(sl[1]);
// prints 1
System.Console.WriteLine(sl.GetByIndex(0));
// prints 1
```

SortedList

- Iterate through SortedList
  - Using index

```
for(int i=0;i< sl.count;i++)
{
  Console.WriteLine($"key={sl.GetKey(i)}\t value={sl.GetByIndex(i)}");
}
```

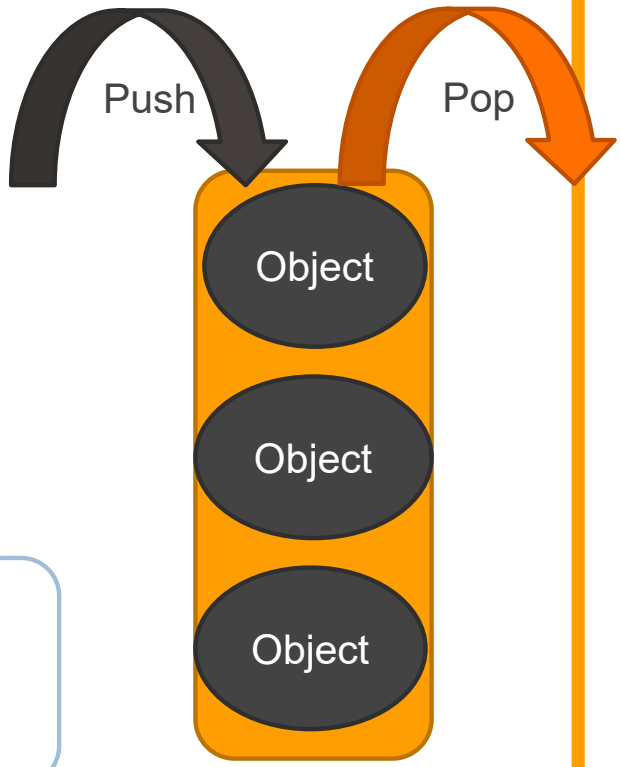
  - Using DictionaryEntry

```
 foreach(DictionaryEntry pair in sl)
{
  Console. WriteLine($"key={pair.Key}\t value={pair.Value}");
}
```

Stack

```
public class Stack : ICloneable, System.Collections.ICollection
```

- Represents a simple last-in-first-out (LIFO) non-generic collection of objects.
- Methods
  - Push()
  - Pop()
  - Peek ()
  - Clear()
  - Contains(Object)
  - Object[] ToArray()
- Properties
  - Capacity
  - Count

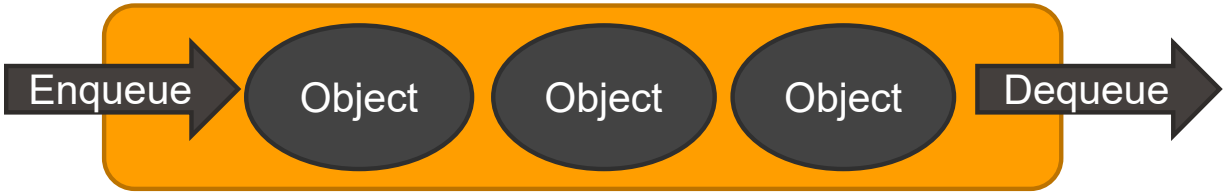



```
Stack s=new Stack();
s.Push(1);
s.Push(2);
Console.WriteLine(s.Pop());// prints 2
```

Queue

```
public class Queue : ICloneable, System.Collections.ICollection
```

- Represents a first-in, first-out collection of objects.
- Methods
  - Enqueue()
  - Dequeue()
  - Peek ()
  - Clear()
  - Contains(Object)
  - Object[] ToArray()

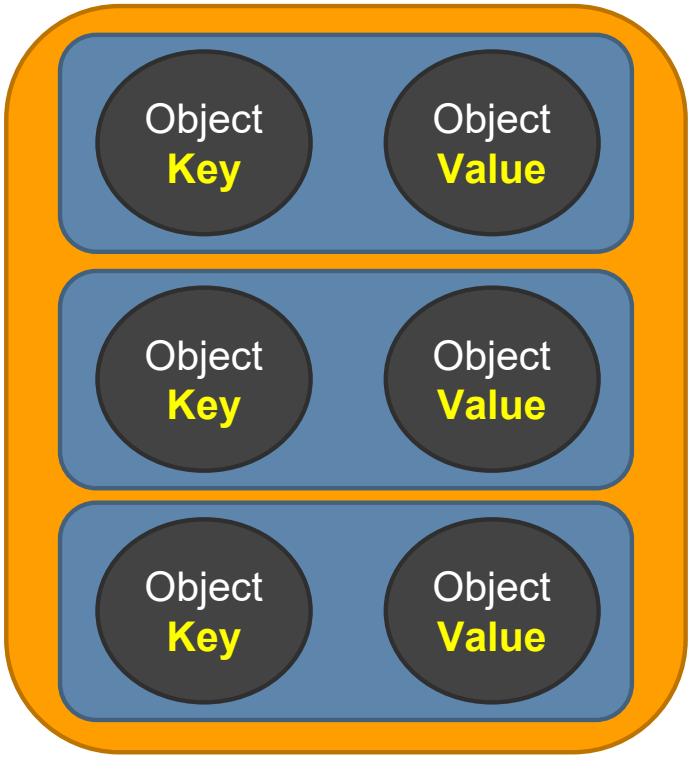


- Properties
  - Capacity
  - Count

```
Queue q = new Queue();
q.Enqueue(1);
q.Enqueue(2);
Console.WriteLine(q.Dequeue().ToString()); // prints 1
```

Hashtable

- Store Data in Key-value format, where keys are unique and used in indexer
  - Ex: Dictionary (word – meaning)
- Methods
  - void Add(object key, object value)
  - void Clear()
  - bool ContainsKey(object key)
  - bool ContainsValue(object value)
  - void Remove(object key);



Hashtable

- Properties
  - Count
  - Item[Key]
  - Keys
  - values

```
Hashtable ht = new Hashtable();
ht.Add("One",1 );
ht.Add("Two", 2);
ht.Add("three", 3);
Console.WriteLine(ht["three"].ToString()); // print 3
```

```
 foreach(DictionaryEntry node in ht)
{
   Console.WriteLine(node.ToString()); // print 3
}
```

```
foreach (var k in ht.Keys)
{
   Console.WriteLine(k.ToString());
}
```

Assignment

- Modify Menu program to use ArrayList instead of Array of Employees
  - New
    - Add one employee at a time
  - Display
    - Display all Employees
  - Search
    - Search employee by (Id , name)
  - Sort
    - Sort Employee using Sort(IComparer)

Generics

Generics

- Generics considered as *template* with *placeholder* for type

```
static void Swap (ref int x, ref int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

```
static void Swap (ref char x, ref char y)
{
        char temp;
        temp = x;
        x = y;
        y = temp;
}
```

Generic Method

- Definition

```
static void Swap <T> (ref T x, ref T y)
{
        T temp;
        temp = x;
        x = y;
        y = temp;
}
```

- Calling

```
Swap <char> (ref x, ref y);
```

```
Swap (ref x, ref y);
```

```
Swap <int> (ref x, ref y);
```

Default generic value

- *default(T)*
  - Ex : return of pop Method

```
public T pop()
{
  if (tos > 0)
  {
    tos--;
    return stk[tos];
  }
  else
    return default(T);
}
```

```
public int pop()
{
  if (tos > 0)
  {
   tos--;
   return stk[tos];
  }
  else
    return -1;
}
```

Generic Class

- Definition

Generic type

```
public class Demo <T>
{
        public T v;
        public Demo(T x )
        { v=x;}
}
```

```
public class Pair <T,U>
{
        public T v1;
        public U v2;
        public Pair(T x,U y )
        { v1=x; v2=y; }
}
```

- Declare Reference and Instantiating an Object

Constructed type

```
Demo<int> D=new Demo<int>(10);
```

```
Pair<int,string> D2=new Demo <int,string>(10,"Hi");
```

Generic Interface

- Definition

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

Generic Constraint

- Arithmetic operation Constraint

```
class Complex<T>
{
    public T real;
    public T img;
    public Complex()
    {
        real = img = default;
    }
    public static Complex<T> operator +(Complex<T> c1, Complex<T> c2)
    {
        Complex<T> c = new Complex<T>();
        c.real = c1.real + c2.real; // Error cant apply operator + for T and T
    }
}
```

Generic Type Constraint

- Constraint on T could be achieve using *where* statement

```
GenericTypeName<T> where T   : contraint1, constraint2
```

```
class GenericClass<T, U> where T : class1, Interface1
                         where U : new()
    {...}
```

Generic Type Constraint

class	The type argument must be any class, interface, delegate, or array type.
class?	The type argument must be a nullable or non-nullable class, interface, delegate, or array type.
struct	The type argument must be non-nullable value types such as primitive data types int, char, bool, float, etc.
new()	The type argument must be a reference type which has a public parameterless constructor. It cannot be combined with struct and unmanaged constraints.
notnull	Available C# 8.0 onwards. The type argument can be non-nullable reference types or value types. If not, then the compiler generates a warning instead of an error.
unmanaged	The type argument must be non-nullable unmanged types.

Generic Type Constraint

base class name	The type argument must be or derive from the specified base class. The Object, Array, ValueType classes are disallowed as a base class constraint. The Enum, Delegate, MulticastDelegate are disallowed as base class constraint before C# 7.3.
<base class name>?	The type argument must be or derive from the specified nullable or non-nullable base class
<interface name>	The type argument must be or implement the specified interface.
<interface name>?	The type argument must be or implement the specified interface. It may be a nullable reference type, a non-nullable reference type, or a value type
where T: U	The type argument supplied for T must be or derive from the argument supplied for U.

Generic Type Constraint

INumber<T>	The type argument must be numeric type
IBinaryInteger<T>	The type argument must be integer

Generic and Inheritance

- Inheriting generic types

```
public class GenStack <T>
{
        public T  [ ] stk;
        public int  size;
}
```

```
class specialStack <T>:Genstack<T>
{
   …
}
```

```
class specialStack:Genstack<int>
{
   …
}
```

Generic and Inheritance

- Implementing Generic Interface

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

```
public class GenClass2<T>:
IGenInteface<T>
    {
        T t1;
        public T Prperty
        {
            get
            {
                return t1;
            }
            set
            {
                t1 = value;
            }
        }
    }
```

```
 public class Class3 :
 IGenInteface<int>
    {
        int t2;
        public int Prperty
        {
            get
            {
                return t2;
            }
            set
            {
                t2 = value;
            }
        }
    }
```

HasValue

Nullable value Type

- A nullable value type allows a variable to contain either a value or null
- Declaration and assignment

```
string s=null;
int z=null; // error
Nullable<int> c=null;
int? k=null;
```

- HasValue property
- Value Property

```
int? b = 10;
if (b.HasValue)
{
  Console.WriteLine($"b is {b.Value}");
}
else
{
  Console.WriteLine("b does not have a value");
}
// Output: "b is 10"
```

Nullable Reference Type

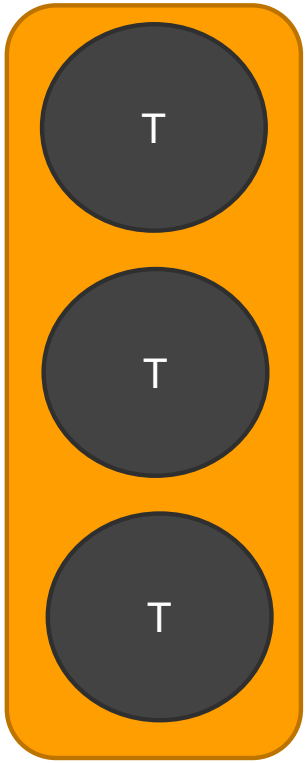
- Eliminate warning generated to help developer to find potential null reference error (Exception)

```
string s=null; // warning
String? z=null;
```

Generic Collection

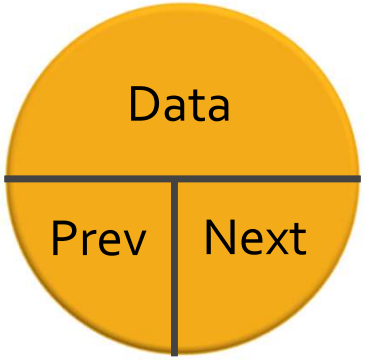
- It ensure Type safe (strongly typed)
- *System.Collections.Generic* namespace
- List<T>
- Stack<T>
- Queue<T>
- SortedList<TKey,Tvalue >
- Dictionary<TKey,TValue>

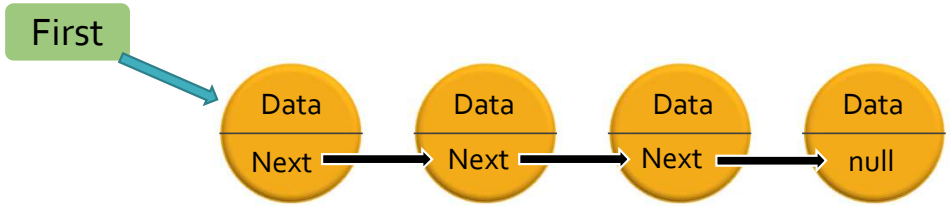
```
List<int> l = new List<int>();
List<employee> empl = new List<employee>();
```



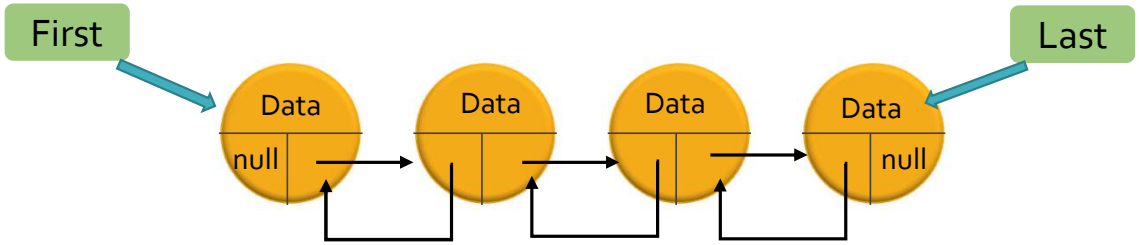
Generic Collection

- LinkedList





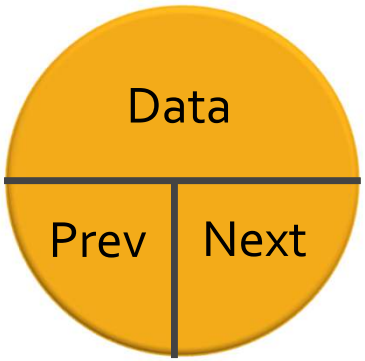
Single Linked List



Double Linked List

LinkedList

- **LinkedListNode<T>** Class
  - Properties
    - List
    - Next
    - Previous
    - Value
    - ValueRef
  - Methods
    - LinkedListNode (T value) => constructor



LinkedList

- LinkedList<T> class
  - Double Linked List
- Properties
  - Count
  - First
  - Last
- Methods
  - AddAfter
  - AddBefore
  - AddFirst
  - AddLast
  - Find
  - FindLast
  - Remove
  - RemoveFirst
  - RemoveLast

Collection Initializers

```
List<string> l;
l = new List<string> { "Ahmed", "Aly", "Mohamed" };
```

- Dictionary

```
var Numbers2 = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

Assignment

- Write a generic stack class that implmenet Generic interface contain only one Method => T GetByIndex(int index);
- Change *ArrayList* To *List<T>* in menu program
  - Change sort (by name , by ID , by Salary) using Icomparer<T> interface