Generic Collections

Arrays

```
[System.Runtime.InteropServices.ComVisible(true)]
[System.Serializable]
public abstract class Array : ICloneable, System.Collections.IList,
                              System.Collections.IStructuralComparable,
                              System.Collections.IStructuralEquatable
```

Provides methods:

- for creating, manipulating, searching, and sorting arrays
- thereby serving as the **base** class for all arrays in the common language runtime

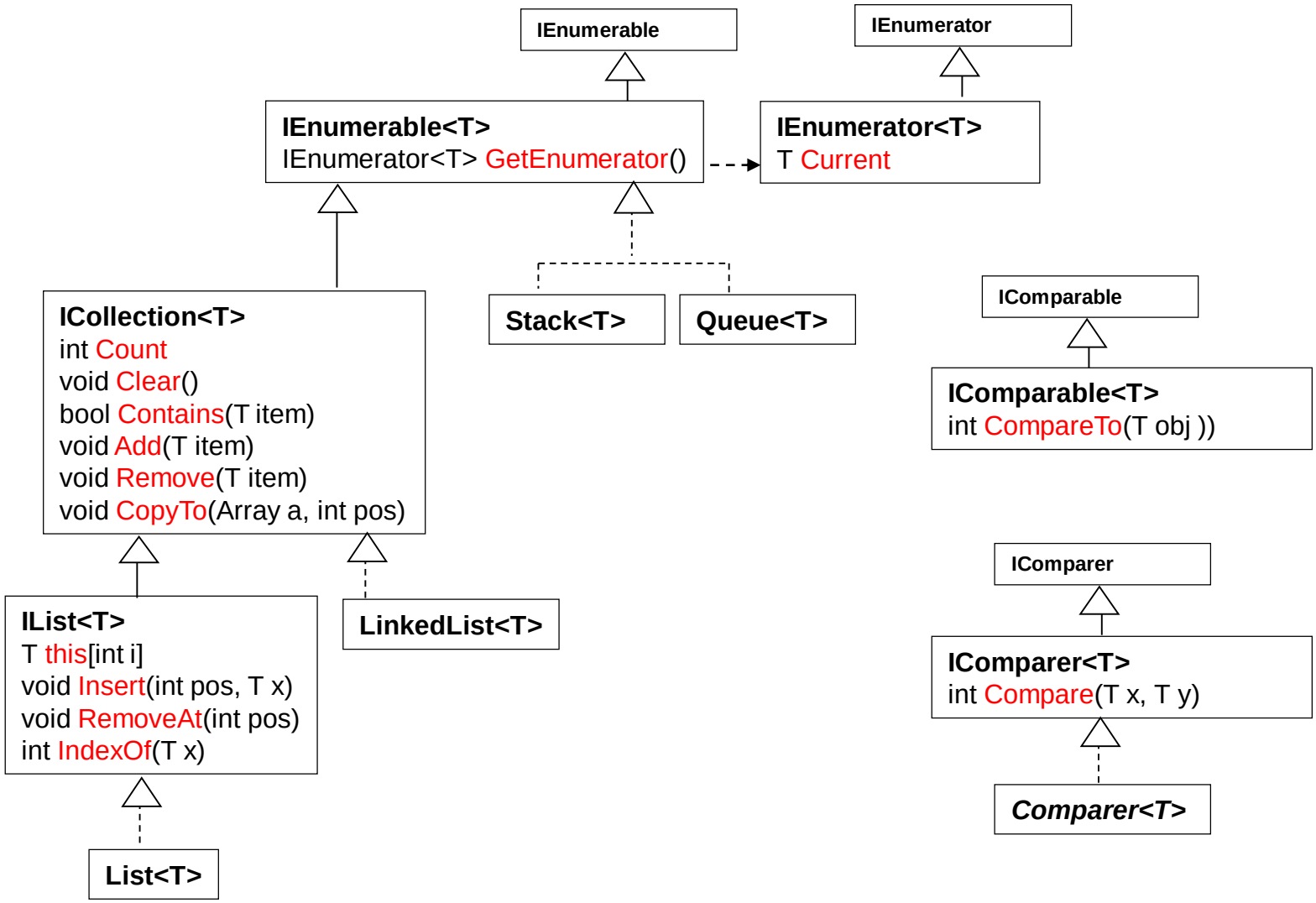
Arrays

- An array can be Single-Dimensional, Multidimensional or Jagged. An array can have a maximum of 32 dimensions.
- The number of dimensions and the length of each dimension are established when the array instance is created.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays
- Array elements can be of any type, including an array type.

```
var multiDim = new int[2, 3, 3];

for (uint i = 0; i <= multiDim.GetUpperBound(0); ++i)
  for (uint j = 0; j <= multiDim.GetUpperBound(1); ++j)
    for (uint k = 0; k <= multiDim.GetUpperBound(2); ++k)
         multiDim[i, j, k] = (int)(i + j + k);

foreach (var e in multiDim)
         Console.WriteLine($"elements: {e}");

// Creates and initializes a new three-dimensional Array of type Int32.
Array myArr = Array.CreateInstance(typeof(Int32), 2, 3, 4);
```

Generic Collections

Namespace **System.Collections.Generic** provides generic classes for collections



IEnumerable<T> & IEnumerator<T>

- IEnumerable<T> for anything which is enumerable

```
interface IEnumerable<T> : IEnumerable {
    IEnumerator<T> GetEnumerator();
}
```

- IEnumerator<T> realizes an iterator

```
interface IEnumerator<T> : IEnumerator {
    T Current {get;}
    bool MoveNext();
    void Reset();
}
```

ICollection<T>

- Base interface for collections

```
interface ICollection<T>:
                    IEnumarable<T> {
    //---- Properties
    int Count {get;}
    bool IsReadOnly {get;}

    //---- Methods
    void Add(T elem);
    bool Remove(T elem);

    void Clear();
    bool Contains(T elem);
    void CopyTo(T[] a, int index);
}
```

- number of elements
- read only?

- adding an element
- removing an element

- remove all
- containment
- copies elements into array a (beginning at position index)

LinkedList<T>

- Linked list implementation of ICollection<T>
- works with LinkedListNode<T>

```
public class sealed  LinkedListNode<T> {
   public T Value { get; set; }
   public LinkedListNode<T> Next { get; }
   public LinkedListNode<T> Previous { get; }
}
```

```
public class LinkedList<T>: ICollection<T> {
    //---- Properties
    …
    public LinkedListNode<T> First { get; }
    public LinkedListNode<T> Last { get; }

    //---- Methods
    public LinkedListNode<T> AddFirst ( T value)
    public LinkedListNode<T> AddLast ( T value)
    public LinkedListNode<T> AddAfter (
                LinkedListNode<T> node, T value )
    …

}
```

- first node
- last node

- add first node with value
- add last node with value
- add new node after node

IList<T>

- Interface for collections with positioned access

```
interface IList<T>: ICollection<T>: {

    //---- Properties
    T this[int index] {get; set;}

    //---- Methods
    void Insert(int index, T elem);
    bool RemoveAt(int index);
    int IndexOf(T elem);
}
```

- Indexer for direct access
- adding an element at position index
- removing an element at position index
- Position of element elem

List<T>

- Standard implementation of IList<T>

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T> {
    // IEnumerable<T>: GetEnumerator
    // ICollection<T>: Count, CopyTo, Add, Contains, ...
    // IList<T>: Insert, RemoveAt, IndexOf, ...

    //----- Constructors
    public List();
    public List(IEnumerable<T> collection);
    public List(int capacity);

    //----- Properties
    virtual int Capacity {get; set;}
    public int Count { get; }
    public T this [int index] { get; set; }
    …
```

- properties and methods of implemented interfaces
- constructors
- reserved space in list
- number of elements
- indexer for positioned access

List<T>

```
//----- Methods
public virtual IList<T> GetRange(int index, int count);
public virtual void AddRange(IEnumerable<T> c);
public virtual void InsertRange(int i, IEnumerable<T> c);
public virtual void RemoveRange(int index, int count);
public virtual int LastIndexOf(T e);

public virtual int BinarySearch(T e);
public virtual int BinarySearch(T e, IComparer<T>);

public virtual void Sort();
public virtual void Reverse();
public virtual T[] ToArray();
public virtual void TrimExcess();

}
```

- subset
- adding a set of elements
- inserting a set of elements
- removing a set of elements
- last position where e occurs

- binary serach for e
- binary search with IComparer

- sorting
- inversion of elements
- copying elements into T[] array
- setting capacity to current number of elements

IComparable<T> & IComparer<T>

- IComparable<T> is interface for types with order

```
public interface IComparable<T> {
    int CompareTo(T obj); // -1 if x < y, 0 if x == y, 1 if x > y
}
```

- IComparer<T> is interface for realizing comparison objects

```
public interface IComparer <T> {
    int Compare(T x, T y);  // -1 if x < y, 0 if x == y, 1 if x > y
}
```

IComparer<T> Example

```
public class PersonComparer<T> : IComparer<T> where T: Person {
        public int Compare(T person1, T person2)
        {
            return person1.ssNr.CompareTo(person2.ssNr);
        }
}
```

```
    var persons = new Person[]
    {
          new Person("030819778345", "Herbert Miller"),
          new Person("010519506534", "Mary Master"),
          new Person("100719654298", “Harry Monster")
    };

    Array.Sort(persons, new PersonComparer<Person>());

    foreach (Person p in persons) {
        Console.WriteLine(p.ToString());
    }
```

```
010519506534, Mary Master
030819778345, Herbert Miller
100719654298, Harry Monster
```

List<T> Example

```
using System;
using System.Collections.Generic;
...
var list = new List<Person>
{
      new Person("030819778345", "Herbert Miller"));
      new Person("010519506534", "Mary Master"));
}
…
list.Add(new Person("100719654298", "Harry Monster"));

list.Sort(new PersonComparer<Person>());
// foreach (var p in list) Console.WriteLine(p);
list.ForEach( p => Console.WriteLine(p));
```

Output:

```
010519506534, Mary Master
030819778345, Herbert Miller
100719654298, Harry Monster
```

```
list.Reverse();
for (int i = 0; i < list.Count; i++) Console.WriteLine(list[i]);
```

```
100719654298, Harry Monster
030819778345, Herbert Miller
010519506534, Mary Master
```

List<T> Example

```
using System;
using System.Collections.Generic;
...
var list = new List<string>();
list.Add("Anton"); list.Add("Dora"); list.Add("Berta");
list.Add("Emil"); list.Add("Caesar");

list.Sort();

var i = list.BinarySearch("Emil");
Console.WriteLine("Pos. {i}: {list[i]}");
```

```
Pos. 4: Emil
```

```
//----- Conversion to static array
var arr = list.ToArray();
foreach (var s in arr) Console.WriteLine(s);
```

```
Anton
Berta
...
```

Queue<T>

- Queue<T> realizes buffer with FIFO strategy

```
public class Queue<T> : ICollection, IEnumerable<T> {

    // IEnumerable<T>: GetEnumerator
    // ICollection<T>: Count, CopyTo, ...

    public Queue();
    public Queue(IEnumerable<T> c);
    public Queue(int capacity);

    //----- Methods
    public virtual void Enqueue(T elem);
    public virtual T Dequeue();
    public virtual T Peek();
    …
}
```

- implementation of interfaces ICollection and IEnumerable
- constructors
- appending element in the back
- removing first element
- accessing first element without removing it

Queue<T> Example

```
using System;
using System.Collections.Generic;
...
var q = new Queue<string>();
q.Enqueue("Anton"); q.Enqueue("Berta");
q.Enqueue("Caesar"); q.Enqueue("Dora");
while (q.Count > 0) Console.Write(q.Dequeue());
```

```
Anton Berta Caesar Dora
```

Stack<T>

- Stack<T> realizes generic stack with LIFO strategy

```
public class Stack<T> : ICollection, IEnumerable<T> {

    // IEnumerable<T>: GetEnumerator
    // ICollection<T>: Count, CopyTo, ...

    //----- Constructors
    public Stack();
    public Stack(IEnumerable<T> c);
    public Stack(int capacity);

    //----- Methods
    public virtual void Push(T elem);
    public virtual T Pop();
    public virtual T Peek();
    …
}
```

- implementation of interfaces ICollection und IEnumerable
- constructors
- putting element on the stack
- removing topmost element
- reading topmost element without removing it

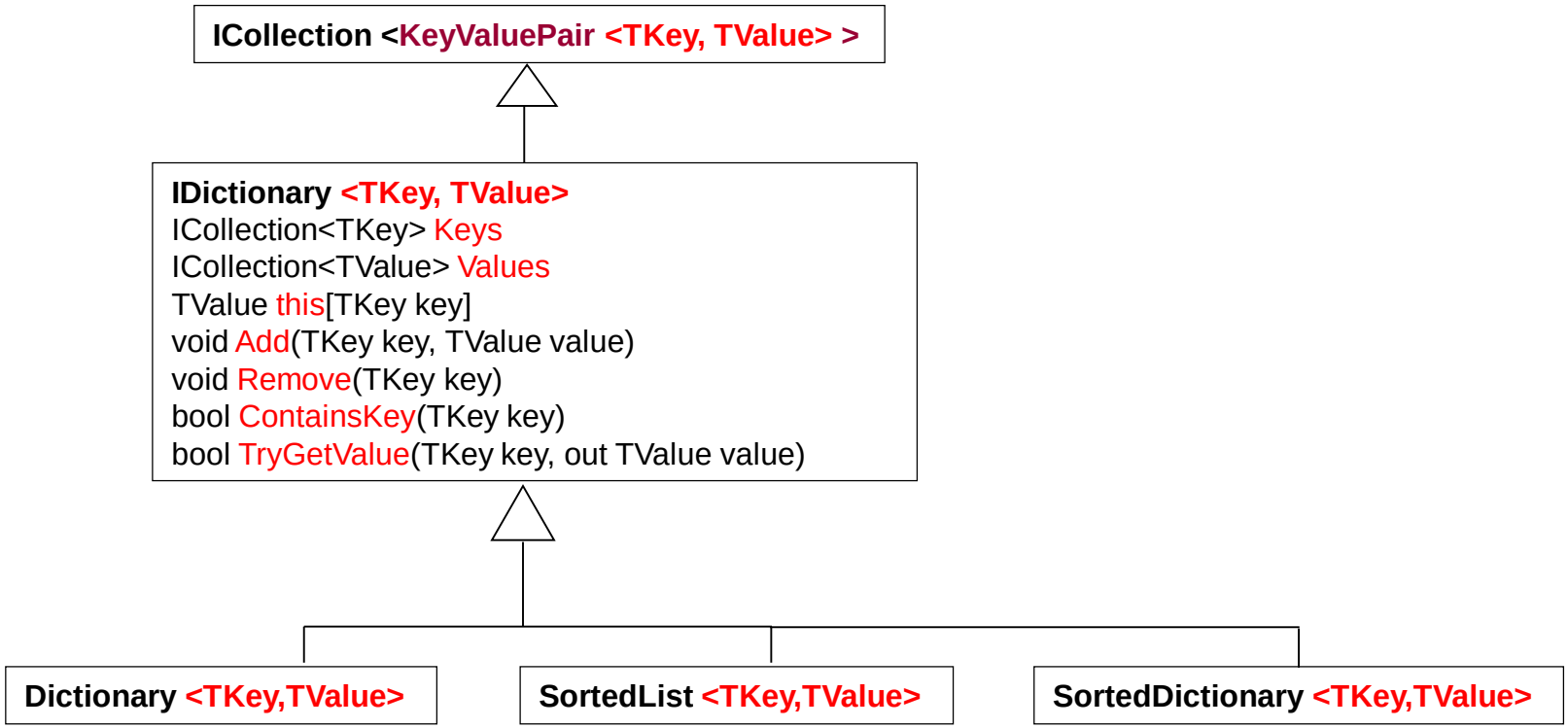
Stack<T> Example

```
var s = new Stack<string>();

s.Push("Anton");
s.Push("Berta");
s.Push("Caesar");
s.Push("Dora");

while (s.Count > 0) Console.Write(s.Pop());
```

Dora Caesar Berta Anton

Generic Dictionary Classes

- Generic types for mappings from keys to values



- Implementation with hashtable
- Retrieving a value by using its key is very fast, close to O(1)

- Implementation as an array of key/value pairs, sorted by the key
- O(log n) **retrieval**
- O(n) **insertion** and **removal**

- Implementation as a binary tree of key/value pairs
- O(log n) **retrieval**
- O(log n) **insertion** and **removal** for unsorted data

IDictionary<TKey, TValue>

- General interface for mappings from keys to values

```
public struct KeyValuePair<TKey, TValue> {
   public KeyValuePair(TKey key, TValue value);
   public TKey Key {get; }
   public TValue Value {get; }
}
```

```
interface IDictionary<TKey, TValue>:
        ICollection<KeyValuePair<TKey, TValue>>,
        IEnumerable<KeyValuePair<TKey, TValue>> {

   // inherits from ICollection<T>: Count, CopyTo, ...

   //----- Properties
   ICollection<TKey> Keys {get;}
   ICollection<TValue> Values {get;}
   TValue this[TKey key] {get; set;}

   //----- Methods
   void Add(TKey key, TValue value);
   void Remove(TKey key);
   bool ContainsKey(TKey key);
   bool TryGetValue(TKey key, out TValue value);
}
```

Access to:

- set of keys
- set of values
- value for an key

- adding a key-value pair
- Removing a value for a key
- Checking if value for key contained
- trial to access value for an key; returns **false** if unsuccessful and **value** gets the appropriate default value

Dictionary<TKey, TValue>

- Implementation of IDictionary<TKey, TValue> with hashtable

```
public class Dictionary<TKey, TValue> :
        IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>,
        IEnumerable<KeyValuePair<TKey, TValue>>, ... {

    //----- implemented interfaces
    // ICollection: Count, CopyTo, ...
    // IDictionary: Clear, Add, Remove, Contains, GetEnumerator, Indexer, ...

    //----- Constructors
    public Dictionary();
    public Dictionary(int capacity);
    public Dictionary(IEqualityComparer<TKey> comparer);
    public Dictionary(IDictionary<TKey, TValue> d);

    //----- Methods
    public virtual bool ContainsKey(TKey key);
    public virtual bool ContainsValue(TValue val);
    ...
}
```

Dictionary<TKey, TValue> Example

- Dictionary with SSN as keys and Person-objects as values

```
var tab = new Dictionary<long, Person>();

tab.Add(3181030750, new Person("3181030750", "Mike Miller"));
tab.Add(1245010770, new Person("1245010770", "Susanne Parker"));
tab.Add(2345020588, new Person("2345020588", "Roland, Howard"));
tab.Add(1245300881, new Person("1245300881", "Douglas Adams"));

foreach (var e in tab)
    Console.WriteLine(e.Value + ": " + e.Key);
if (tab.ContainsKey(1245010770))
    Console.WriteLine("Person with SSN 1245010770: " + tab[1245010770]);
```

```
3181030750, Mike Miller: 3181030750
1245010770, Susanne Parker: 1245010770
2345020588, Roland, Howard: 2345020588
1245300881, Douglas Adams: 1245300881
Person with SSN 1245010770: 1245010770, Susanne Parker
```

SortedDictionary<TKey, TValue>

- Sorted according to keys
- Implementation with tree

```
public class SortedDictionary<TKey, TValue> :
        IDictionary<TKey, TValue>, ICollection<KeyVauePair<TKey, TValue>>,
        IEnumerable<KeyVauePair<TKey, TValue>>, ... {

    //----- Constructors
    public SortedDictionary();
    public SortedDictionary(IComparer<TKey> c);
    …
    //----- Properties
    public virtual IComparer<TKey> Comparer { get; }
    public virtual TValue this [ TKey ] { get; set; }
    //----- Methods
    public virtual void Add(TKey key, TValue value);  // adds key-value pair
    public virtual void RemoveAt(int i);  // removes key-value pair with position i
    public virtual bool ContainsKey(TKey key); // key contained?
    public virtual bool ContainsValue(TValue val); // value contained?
    public virtual int IndexOfKey(TKey key);  // returns position of key
    public virtual int IndexOfValue(TValue value);  // returns position of value      …
}
```

Example

SortedDictionary<TKey, TValue>

```
var persons = new SortedDictionary<long, Person>();
```

```
persons.Add(3181030750, new Person("Mike", "Miller"));
persons.Add(1245010770, new Person("Susanne", "Parker"));
persons.Add(2345020588, new Person("Roland", "Howard"));
persons.Add(1245300881, new Person("Douglas", "Adams"));
```

```
foreach (var pEntry in persons) {
    long ssn = pEntry.Key;
    Person person = pEntry.Value;
    System.Console.WriteLine("SSN {0} : {1}", ssn, person.ToString());
}
```

```
SSN 1245010770 : Susanne Parker
SSN 1245300881 : Douglas Adams
SSN 2345020588 : Roland Howard
SSN 3181030750 : Mike Miller
```

Generic Set Classes

- Providing interfaces for the abstraction of sets