Arrays

Array in C# is an object:

- **base class** System.Array
- public int Length { get; }

Property Length **returns the number of all the elements in all the dimensions of an array**

1. Declaring Array

<type>[ ] *<array_name>*;

2. Instantiating Array

<array_name> = new *<type>*[*size*];

Example: Single-dimensional array

```
using System;
class SingleArray
{
   private int[] a;
   public SingleArray (int number)
   {
      a = new int [number];
      Console.WriteLine ("Enter {0} integers.",number);
      for (int i = 0; i < number; i++)
      {
         Console.Write ("a[{0}]=", i);
         a[i] = int.Parse (Console.ReadLine());
      }
   }
```

```
   public void PrintArray()
   {
      Console.WriteLine ("Array");
      for (int i = 0; i < a.Length; i++)
         Console.Write (a[i] + " ");
      Console.WriteLine ();
   }
   static void Main()
   {
      SingleArray x = new SingleArray(5);
      x.PrintArray ();
   }
}
```

3. Multidimensional Array

<type>[, , ...,] *<array_name>*;

or

<type>[][]...[] *<array_name>*;

public int GetLength (int *dimension*)

Method Array.GetLength – **determines the length, or upper bound, of each** dimension **of the array.**

public int Rank { get; }

Property Rank **gets the rank (number of dimensions) of the** Array**.**

Example: Two-dimensional array

```
using System;
class TwoDimArray
{
   private float[,] a;
   public TwoDimArray (int number1, int number2)
   {
      a = new float [number1, number2];
      Console.WriteLine ("Enter an matrix {0}x{1} " +
                   "of real numbers.", number1, number2);
      for (int i = 0; i < a.GetLength(0); i++)
         for (int j = 0; j < a.GetLength(1); j++)
         {
            Console.Write ("a[{0},{1}]=", i, j);
            a[i,j] = float.Parse (Console.ReadLine());
         }
   }
```

```
   public void PrintArray()
   {
      Console.WriteLine ("Array");
      for (int i = 0; i < a.GetLength(0); i++)
      {
         for (int j = 0; j < a.GetLength(1); j++)
            Console.Write ("{0,5:f2}", a[i,j]);
         Console.WriteLine ();
      }
   }
   static void Main()
   {
      TwoDimArray x = new TwoDimArray (5, 3);
      x.PrintArray ();
   }
}
```

Example: **Jagged array – an array of arrays**

```
using System;
class JaggedArray
{
   private int[][] a;

   public JaggedArray (int rows, params int[] n)
   {   a = new int [rows][];
       for (int i = 0; i < rows; i++)
          a[i] = new int [n[i]];
       for (int i = 0; i < a.Length; i++)
       {
          Console.WriteLine("Enter {0} integers.", a[i].Length);
          for (int j = 0; j < a[i].Length; j++)
          {   Console.Write ("a[{0},{1}]=", i, j);
              a[i][j] = int.Parse (Console.ReadLine ());
          }
       }
   }
```

```
   public void PrintArray ()
   {
      Console.WriteLine ("Jagged array");
      for (int i = 0; i < a.Length; i++)
      {
         for (int j = 0; j < a[i].Length; j++)
            Console.Write ("{0,5}", a[i][j]);
         Console.WriteLine ();
      }
   }
   static void Main()
   {
       JaggedArray x = new JaggedArray (3, 2, 5, 3);
       x.PrintArray ();
   }
}
```

Indexers

Indexer – **treats an object like an array; known as a smart array.**

1. **Defining Indexer**
   - **takes an index argument**
   - this **keyword is used as the name of the indexer**

```
class <class_name>
{
   public <type> this [<index_type> index]
   {
      get  {   // Return desired data            }
      set  {   // Set desired data               }
   }
}
```

2. **Instantiating Indexer – create an object of this class and treat the object as an array:**

<class_name> <object_name> = new *<class_name>*();

<object_name> [*index*] = *<any_object>*;

Example: **Array of strings (list of strings)**

Class System.Collections.ArrayList **– stores a collection of objects (dynamic array of objects).**

Method ArrayList.Add **– adds an object at the end of the collection.**

Property ArrayList.Count **– returns the current number of elements in the collection.**

```
using System;
using System.Collections;
class MyListBox
{
   protected ArrayList data = new ArrayList();

   public object this[int idx]
   {
        get
        {    if (idx>-1 && idx < data.Count)
                return data[idx];
             else
                throw new InvalidOperationException
                                 ("The index is out of range");
        }
```

```
        set
        {    if (idx >-1 && idx < data.Count)
                data[idx] = value;
             else if (idx == data.Count)
                data.Add (value);
             else
                throw new InvalidOperationException
                                 ("The index is out of range");
        }
   }
}
class IndexersApp
{   static void Main()
    {   MyListBox list = new MyListBox();
        list[0] = "George"; list[1] = "group 80"; list[2] = "FCSC";
        Console.WriteLine ("{0}, {1}, {2}", list[0], list[1], list[2]);
    }
}
```

Example: Chess-board 8x8 indexer with two parameters: row – A ÷ H, column – 1 ÷ 8

```
using System;
class Grid
{
   const int NumRows = 8;
   const int NumCols   = 8;
   string[,] cells = new string[NumRows, NumCols+1];
```

```
public string this[char c, int col]
{
    get
    {
        c = Char.ToUpper (c);
        if (c < 'A' || c > 'H')
        {
            throw new ArgumentException();
         }
         if (col < 1 || col > NumCols)
         {
             throw new IndexOutOfRangeException();
         }
         return cells[c - 'A', col];
      }
```

```
        set
        {
          c = Char.ToUpper(c);
          if (c < 'A' || c > 'H')
          {
              throw new ArgumentException();
          }
          if (col < 1 || col > NumCols)
          {
              throw new IndexOutOfRangeException();
          }
          cells[c - 'A', col] = value;
        }
    }
}
```

```
class Test
{
   static void Main()
   {
       Grid g = new Grid();
        g['A',1] = "castle";
        g['H',8] = "pawn";
   }
}
```

String Handling

String – a sequence of characters.

Class System.String (alias string) – represents an immutable string of characters (its value can't be modified once it's been created).

String handling classes:

- StringBuilder
- StringFormat
- StringCollection
- **and so on**

Example:

```
public string Replace (char oldChar, char newChar);
public string Replace (string oldValue, string newValue);
public string Insert (int startIndex, string value);
public string ToUpper();
using System;
class TestStringApp
{   static void Main (string[] args)
    {   string a = "strong";
        // Replace all 'o' with 'i'
        string b = a.Replace ('o', 'i');
        Console.WriteLine (b);                // string
        string c = b.Insert (3, "ipl");       // stripling
        string d = c.ToUpper ();
        Console.WriteLine (d);                // STRIPLING
    }
}
```

Example:

```
public int CompareTo (string strB);
// String comparison (method CompareTo) and
// overloaded operator ==
if (d==c)                    // or if (d.CompareTo (c) == 0)
  Console.WriteLine ("same");
else
  Console.WriteLine ("different");
```

Results:

```
different
```

Example:

```
// Replace doesn’t change the string.
// Reassigning a string variable.
string q = "strong";
q = q.Replace ('o', 'i');
Console.WriteLine (q);
```

Example:

```
public string Substring (int startIndex);
public string Substring (int startIndex, int length);
// String concatenation and finding a substring
string e = "Hi"+", ";
e += "guy";
Console.WriteLine (e);                    // Hi, guy
string f = e.Substring (1,4);
Console.WriteLine (f);                    // i, g
for (int i = 0; i < f.Length; i++)
    Console.Write("{0,-3}", f[i]);
Console.WriteLine();                      // i  ,    g
```

Example:

```
public bool StartsWith (string value);
public string Remove (int startIndex, int count);
string g = null;
if (e.StartsWith ("Hi"))
    g = e.Remove(2,1);
Console.WriteLine (g);              // Hi,guy
```

Example:

```
public static string Format (string format, params object[] args);
int x = 16;
decimal y = 3.57m;
string h = String.Format("Item {0} sells at {1:C}", x, y);
Console.WriteLine (h);       // Item 16 sells at 3,57 лв
```

Example:

```
// Concatenate a string with any other type using the plus sign +
// (all types inherit object.ToString)
string t = "Item "+12+" sells at "+3.45+" lv";
Console.WriteLine (t);
```

Results:

```
Item 12 sells at 3,45 lv
```

Example:

```
// String.Format and Console.WriteLine take as the last argument
// params object[]
Console.WriteLine("Hi, {0} {1} {2} {3} {4} {5} {6} {7} {8}",
    123, 45.67, true, 'A', 4, 5, 6, 7, '8');
string u=String.Format("Hi, {0} {1} {2} {3} {4} {5} {6} {7} {8}",
    123, 45.67, true, 'A', 4, 5, 6, 7, '8');
Console.WriteLine (u);
```

Results:

```
Hi, 123 45,67 True A 4 5 6 7 8
Hi, 123 45,67 True A 4 5 6 7 8
```

Example:

```
public string[] Split (params char[] separator);
char[] separator = new char[]{‘ ‘,’,’,’.’,’:’,’/’,’\\’,’\”};
// The method String.Split splits a string into substrings by
// separator characters
string str = "Documents and Settings";
char[] seps=new char[]{' '};
foreach (string ss in str.Split (seps))
    Console.WriteLine (ss);
```

Results:

```
Documents
and
Settings
```

Disadvantage: Split **is not useful if the substrings are separated by multiple instances of some character (if there are more than one space in the string the result is an empty line between the words).**

Class System.Text.StringBuilder **– represents a mutable string of characters; cannot be inherited.**

Example:

```
public String Builder Append (string value);
public StringBuilder AppendFormat (string format,
                                  params object[] args);
using System.Text;
StringBuilder sb =new StringBuilder("strong");   // Strong
sb.Replace ('o', 'i');                           // String
sb.Insert (3, "ipl");                            // Stripling
sb.Append (" receives");                 // Stripling receives
sb.AppendFormat (",{0}:{1}", 123, 45.6789);
                          // Stripling receives,123:45.6789
sb.Remove (sb.Length-3, 3);
                          // Stripling receives,123:45.6
Console.WriteLine (sb.ToString().ToUpper());
                          // STRIPLING RECEIVES,123:45.6
Console.WriteLine (sb.ToString().ToLower());
                          // stripling reseives,123:45.6
```

Enumerated Type enum

Enumerated type – an order list of names, which can be used in the program and print out.

1. Defining Enumerated Type

<modifier> enum *<type_name>* {*identifier*$_0$, ..., *identifier*$_N$}

identifier$_0$ **has a value** 0**.**

We can write:

identifier$_i$ = *value*;

2. Input

<enumerated_type_variable> =
(*<type_name>*) Enum.Parse (typeof(*<type_name>*), *<string>*);

The *string* **is translated to a value in the** enum **type; an invalid** *string* **will cause** System.ArgumentException**.**

3. Output

<enumerated_type_variable>.ToString ()

4. Assignment

<enumerated_type_variable> = *<type_name>*.*identifier*$_i$;

Example:

```
using System;
class EnumClass
{   enum Season {Spring, Summer, Autumn, Winter};
    static void Main(string[] args)
    {   Season s=(Season)Enum.Parse(typeof(Season),"Winter");
        Console.WriteLine ("The season is {0}.", s);
        Season first = Season.Spring;
        Console.WriteLine ("The first season is {0}.", first);
        Console.WriteLine ("The season {0} has a number {1}.",
                              first, (int)first);
    }
}
```

Results:

```
The season is Winter.
The first season is Spring.
The season Spring has a number 0.
```

Interfaces

Interface – contract; characterizes the behaviors of classes, irrespective of the class hierarchy.

1. Declaring Interfaces

```
[attributes] [modifiers] interface <interface_name>: <interface_list>
{
    // Method declaration
    // Property declarations
    // Indexer declarations
    // Event declarations
}
```

2. Implementing Interfaces – the class <u>must define each and every member of the interface</u>.

Example: Polymorphism via interfaces

```
using System;
interface ISpeaker
{
    void Speak();
}
class Philosopher : ISpeaker
{
   private string philosophy;
    public Philosopher (string thoughts)
    {   philosophy = thoughts;   }
    public void Speak()
    {   Console.WriteLine (philosophy);   }
    public void Pontificate()
    {   for (int i = 1; i <= 3; i++)
          Console.WriteLine (philosophy);
    }
}
```

```
class Dog : ISpeaker
{   public void Speak()
    {   Console.WriteLine ("Bow-wow!");   }
}
class Talking
{   static void Main(string[] args)
    {   ISpeaker current;
        current = new Dog();
        current.Speak();
        current = new Philosopher("I think, hence I exist.");
        current.Speak();
        ((Philosopher)current).Pontificate();
    }
}
```

Results:

```
Bow-wow!
I think, hence I exist.
I think, hence I exist.
I think, hence I exist.
I think, hence I exist.
```

3. Querying for Implementation

- **By using the is operator – checks at run time whether one type is compatible with another type**

  <expression> is *<type>*
  <expression> **– a reference type**

  The value of the expression is true, if:
  <expression> **≠ null and**
  <expression> **can be converted to the** *<type>*
  otherwise is false

```
Dog dog = new Dog();
if (dog is ISpeaker)
   Console.WriteLine ("Implements the interface ISpeaker");
```

- **By using the as operator – converts between compatible types;**

  the result is stored in a local variable and the code can verify whether the variable has a valid value.

  <object> = *<expression>* as *<type>*
  <expression>, *<type>* **– a reference type**

  This is equivalent to
  <expression> is *<type>* ? (*<type>*)*<expression>* : (*<type>*)null

```
ISpeaker speaker;
speaker = dog as ISpeaker;
if (null != speaker)
   Console.WriteLine("Implements the interface ISpeaker");
else
   Console.WriteLine("Doesn't implement the interface ISpeaker");
```

4. Name hiding with interface

- **The common way to call the implemented method from an interface is to cast an instance of the class to the interface type and then call the method.**

```
Dog dog = new Dog();
((ISpeaker)dog).Speak();
```

Call implemented method with a cast

- **The second ability is to call the implemented method without a cast because the interface methods are public.**

```
dog.Speak();
```

Call implemented method without a cast

This way is not desirable when a class implements several interfaces with numerous members.

- **The name hiding technique is used to prevent the implemented members of interfaces becoming public members of the class:**
  - **Remove the public access modifier from the implemented interface member.**
  - **Qualify the implemented member name with the interface name.**

```
class Dog : ISpeaker
{
   void ISpeaker.Speak()
   {   Console.WriteLine ("Bow-wow!");   }
}
…
Dog dog = new Dog();
dog.Speak();
((ISpeaker)dog).Speak();
```

Remove the public modifier and qualify the member name

Error – Speak() is not a public method

OK – call implemented method with a cast

- **Avoiding name ambiguity**

  Implementing multiple interfaces that contain the same member names can result a name collision. IWindow, IFile and MyApplication support closing operation:

```
public interface IWindow
{   void Close();   }
public interface IFile
{   void Close();   }
public class MyApplication : IWindow, IFile
{   public void Close()
    {   Console.WriteLine("Closing the application...");   }
}
…
MyApplication application = new MyApplication();
application.Close();
```

Name collision

Which method Close is called?

```
...
MyApplication application = new MyApplication();
Console.WriteLine("IWindow? {0}", application is IWindow);
Console.WriteLine("IFile? {0}", application is IFile);
application.Close();
```

Results:

```
IWindow? True
IFile? True
Closing the application…
```

Warning!
Problem: The class has implemented one version of Close(), but not both.

The explicit member name qualification is used to avoid name collision:

- **Remove the public access modifier from the implemented interface member.**
- **Qualify the implemented member name with the interface name.**

```
public class MyApplication : IWindow, IFile
{
   void IWindow.Close()
   {   Console.WriteLine("Closing the window...");   }
   void IFile.Close()
   {   Console.WriteLine("Closing the file...");   }
   public void Close()
   {   Console.WriteLine("Closing the application...");   }
}
```

```
...
MyApplication application = new MyApplication();
Console.WriteLine("IFile? {0}", application is IFile);
((IFile)application).Close();
Console.WriteLine("IWindow? {0}", application is IWindow);
((IWindow)application).Close();
application.Close();
```

Results:

```
IFile? True
Closing the file…
IWindow? True
Closing the window…
Closing the application…
```

5. Problems with inheritance and interfaces

- **Class inheriting from a base class and interface that have identical method names**

```
public class Data
{   public void Close()
   {   Console.WriteLine("Closing the data...");   }
}
public interface IWindow
{   void Close();   }
public class MyApplication : Data, IWindow { }
. . .
MyApplication application = new MyApplication();
application.Close();
```

Results:

```
Closing the data…
```

The inherited method Data.Close is called

```
. . .
MyApplication application = new MyApplication();
IWindow window = application as IWindow;
if (null != window)
   window.Close();
```

Results:

```
Closing the data…
```

The inherited method Data.Close is called

- **Derived class has an identical method name as the base class implementation of an interface method**

```
public interface IWindow
{   void Close();   }
public class Data : IWindow
{
   public void Close()
   {   Console.WriteLine("Closing the data...");   }
}
public class MyApplication : Data
{
   public new void Close()
   {   Console.WriteLine("Closing the application...");   }
}
```

```
...
MyApplication application = new MyApplication();
((IWindow)application).Close();
application.Close();
```

Results:

```
Closing the data…
Closing the application…
```

Call IWindow.Close – interface implementation

Call MyApplication.Close – overriden inherited Data.Close (with new)

6. Combining interfaces

Two or more interfaces can be combined so that a class need only implement the combined result.

```
public class Data { }
public interface IWindow
{  void Close(); }
public interface IRead
{  void Read(); }
public interface ICombine : IWindow, IRead
{ }
public class Component : Data, ICombine
{  public void Close()
   {   Console.WriteLine("Closing the component...");   }
   public void Read()
   {   Console.WriteLine("Reading data...");   }
}
```

```
. . .
MyApplication application = new MyApplication();
application.Read();
application.Close();
```

Results:

```
Reading data…
Closing the component…
```

Generics in .NET Framework

Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types they store or use.

```
public class Generic<T>
{
   public T Field;
}
Generic<string> s = new Generic<string>();
s.Field = "A string";
Generic<int> i = new Generic<int>();
i.Field = 123;
```

Advantages

- **Shifts the burden of type safety from user to the compiler.**
- **There is no need to write code to test for the correct data type, because it is enforced at compile time.**
- **The need for type casting and the possibility of run-time errors are reduced.**

Interfaces and Collections in .NET Framework

System.Collections.Generic	System.Collections
IComparer<T>	IComparer
IComparable<T>	System.IComparable
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

IEnumerator

```
// Supports a simple iteration over a nongeneric collection.
public interface IEnumerator
{
   // Gets the current element in the collection.
   object Current { get; }

   // Advances the enumerator to the next element of the collection.
   bool MoveNext ();

   // Sets the enumerator to its initial position, which is before the first element
   // in the collection.
   void Reset ();
}
```

IEnumerator<T>

```
// Supports a simple iteration over a generic collection.
public interface IEnumerator<T> : IDisposable, IEnumerator
{
  // Gets the current element in the collection.
  T Current { get; }
}
```

IEnumerable

```
// Exposes the enumerator, which supports a simple iteration over a non-
// generic collection.
public interface IEnumerable
{
  // Returns an enumerator that iterates through a collection.
  IEnumerator GetEnumerator ();
}
```

IEnumerable<T>

```
// Exposes the enumerator, which supports a simple iteration over a collection
// of a specified type.
public interface IEnumerable<T> : IEnumerable
{
  // Returns an enumerator that iterates through a collection.
  IEnumerator<T> GetEnumerator ()
}
```

IDictionaryEnumerator

```
// Enumerates the elements of a nongeneric dictionary.
public interface IDictionaryEnumerator
{
  // Gets both the key and the value of the current dictionary entry.
  DictionaryEntry Entry { get; }

  // Gets the key of the current dictionary entry.
  object Key { get; }

  // Gets the value of the current dictionary entry.
  object Value { get; }
}
```

ICollection

```
// Defines size, enumerators, and synchronization methods for all nongeneric
// collections.
public interface ICollection : IEnumerable
{
  // Gets the number of elements contained in the ICollection.
  int Count {get; }
  // Gets a value indicating whether access to the ICollection is synchronized
  // (thread safe).
  bool IsSynchronized { get; }
  // Gets an object that can be used to synchronize access to the ICollection.
  object SyncRoot { get; }
  // Copies the elements of the ICollection to an Array, starting at a particular
  // Array index.
  void CopyTo (Array array, int index);
}
```

ICollection<T>

```
// Defines methods to manipulate generic collections.
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  // Gets the number of elements contained in the ICollection.
  int Count { get; }
  // Gets a value indicating whether the ICollection is read-only.
  bool IsReadOnly { get; }
}
```

IList

```
// Represents a non-generic collection of objects that can be individually
// accessed by index.
public interface IList
{
  // Adds an item to the IList.
  void Add (object value);

  // Determines whether the IList contains a specific value.
  bool Contains (object value);

  // Removes the first occurrence of a specific object from the IList.
  void Remove (object value);
…
}
```

IList<T>

```
// Represents a generic collection of objects that can be individually
// accessed by index.
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
   // Gets or sets the element at the specified index.
   T this [int index] { get; set; }

   // Determines the index of a specific item in the IList.
   int IndexOf (T item);

   // Inserts an item to the IList at the specified index.
   void Insert (int index, T item);

   // Removes the IList item at the specified index.
   void RemoveAt (int index);
}
```

IDictionary

```
// Represents a nongeneric collection of key/value pairs.
public interface IDictionary
{
   // Returns an IDictionaryEnumerator object for the IDictionary object.
   IDictionaryEnumerator GetEnumerator ();
   // Adds an element with the provided key and value to the IDictionary
   // object.
   void Add (object key, object value);
   // Determines whether the IDictionary object contains an element with the
   // specified key.
   bool Contains (object key);
   // Removes the element with the specified key from the IDictionary object.
   void Remove (object key);
 …
}
```

IDictionary<TKey,TValue>

```
// Represents a generic collection of key/value pairs.
public interface IDictionary<TKey,TValue> :
   ICollection<KeyValuePair<TKey,TValue>>,
   IEnumerable<KeyValuePair<TKey,TValue>>, IEnumerable
{  // Gets or sets the element with the specified key.
   TValue this [TKey key] { get; set; }
   // Gets an ICollection containing the keys of the IDictionary.
   ICollection<TKey> Keys { get; }
   // Gets an ICollection containing the values in the IDictionary.
   ICollection<TValue> Values { get; }
   // Adds an element with the provided key and value to the IDictionary.
   void Add (TKey key, TValue value);
   // Determines whether the IDictionary contains an element with the
   // specified key.
   bool ContainsKey (TKey key);
   // Removes the element with the specified key from the IDictionary.
   bool Remove (TKey key);
   // Gets the value associated with the specified key.
   bool TryGetValue (TKey key, out TValue value);
}
```

Generic struct KeyValuePair<TKey,TValue> – stores a pair of a generic key and a generic value.

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

IComparer

```
// Exposes a method that compares two objects.
public interface IComparer
{
   // Compares two objects and returns a value indicating whether one is less
   // than, equal to, or greater than the other.
   int Compare (object x, object y);
}
```

IComparer<T>

```
// Exposes a method that compares two objects.
public interface IComparer<T>
{
   // Compares two objects and returns a value indicating whether one is less
   // than, equal to, or greater than the other.
   int Compare (T x, T y);
}
```

IComparable

```
// Defines a generalized comparison method that a value type or class
// implements to create a type-specific comparison method.
public interface IComparable
{
   // Compares the current instance with another object of the same type.
   int CompareTo (object obj);
}
```

IComparable<T>

```
// Defines a generalized comparison method that a value type or class
// implements to create a type-specific comparison method.
public interface IComparable<T>
{
   // Compares the current instance with another object of the same type.
   int CompareTo (T other);
}
```

Example: The collection ArrayList implements the IList interface using an array whose size is dynamically increased as required. The default initial capacity for an ArrayList is 0. As elements are added to a ArrayList, the capacity is automatically increased as required through reallocation. Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

```
using System.Collections;
ArrayList list = new ArrayList();
list.Add ("Software Engineering");
list.Add ("Programming in C#");
IEnumerator myEnumerator = list.GetEnumerator();
while (myEnumerator.MoveNext())
   Console.WriteLine (myEnumerator.Current);
foreach (string element in list)
    Console.WriteLine(element);
```

Example: The collection SortedList implements IDictionary, ICollection, IEnumerable; represents a collection of key/value pairs (DictionaryEntry) that are sorted by the keys and are accessible by key and by index; the elements are sorted by the keys either according to a specific IComparer implementation or according to the IComparable implementation.

```
using System.Collections;
SortedList bookList = new SortedList();
bookList.Add (234567, "Programming in C#");
bookList.Add (123456, "Software Engineering");
IDictionaryEnumerator myEnumerator = bookList.GetEnumerator();
while (myEnumerator.MoveNext())
   Console.WriteLine (myEnumerator.Key + "\t" + myEnumerator.Value);
foreach (DictionaryEntry element in bookList)
    Console.WriteLine(element.Key + "\t" + element.Value);
```

Example: The collection SortedDictionary<TKey,TValue> implements IDictionary<TKey,TValue>, ICollection<KeyValuePair<TKey,TValue>>, IEnumerable<KeyValuePair<TKey,TValue>>, IDictionary, ICollection, and IEnumerable; represents a collection of key/value pairs (KeyValuePair<TKey,TValue>) that are sorted by the keys and are accessible by key and by index

```
using System.Collections.Generic;
SortedDictionary<string, int> phoneBook = new SortedDictionary<string, int>();
phoneBook.Add("Ann", 8871234);
phoneBook.Add("Peter", 8528765);
IEnumerator<KeyValuePair<string, int>> myEnumerator =
                                                phoneBook.GetEnumerator();
while (myEnumerator.MoveNext())
   Console.WriteLine(myEnumerator.Current.Key + "\t" +
                        myEnumerator.Current.Value);
foreach (KeyValuePair<string, int> element in phoneBook)
   Console.WriteLine(element.Key + "\t" + element.Value);
```

Example: Using the IComparable interface with methods for sorting and searching with arrays and collections. The CompareTo method has to be implemented.

```
public int CompareTo (object o);
public static void Sort (Array array, int startindex, int length);
```

```
using System;
class Rational : IComparable
{
   private int numerator, denominator;
   public Rational (int numer, int denom)
   {
      numerator = numer;
      denominator = denom;
      Reduce ();
   }
   private void Reduce()
   {
      int common = Gcd (Math.Abs(numerator), denominator);
      numerator /= common;
      denominator /= common;
   }
```

```
private int Gcd (int n1, int n2)     // Greater common divisor
{
   while (n1 != n2)
      if (n1 > n2)
         n1 -= n2;
      else
         n2 -= n1;
   return n1;
}
public override string ToString()
{
   return numerator + "/" + denominator;
}
```

```
  public int CompareTo (object o)
  {
    Rational op2 = (Rational)o;
    int difference = numerator*op2.denominator-
                     op2.numerator*denominator;
    if (difference < 0)
      return -1;
    else if (difference>0)
      return 1;
    else
      return 0;
    }
}
```

```
class TestRational
{
  static void Main()
  {
    Rational x, y;
    x = new Rational (1, 4);
    y = new Rational (1, 3);
    int flag = x.CompareTo(y);
    if (flag < 0) Console.WriteLine ("x<y");
    else if (flag>0) Console.WriteLine ("x>y");
    else Console.WriteLine ("x=y");
    Rational[] list = new Rational[3] {new Rational (1, 3),
                      new Rational (1, 4), new Rational (2, 5)};
    Array.Sort (list, 0, 3);
    foreach(Rational r in list)
      Console.WriteLine (r);
  }
}
```