Interfaces are closely related to abstract classes where all members are abstract.

A class that implements an interface is required to define all the methods, properties, events, indexers of the interface.

An abstract class is part of a single inheritance subgraph. "is-a" relation.
A car is a vehicle.

An interface can be "mixed in" with many inheritance subgraphs. "implements a" relation.
A car implements boughtWithCredit capability
A house also implement a boughtWithCredit capability

Interfaces are implicitly public
implementing interfaces

To define the interface ADT

```
[access modifiers] interface [I]identifier [:base list]
   { interface method signatures * }
```

interface IScalable {....}  // implicitly public

Stylistically, interfaces are usually named starting with a 'I' to
distinguish them from classes.

base list contains the interface this interface extends

all interface methods are implicitly public -- error if you set access

To implement the interface

```
[access modifiers] class identifier
   [: [base class] [, interface]* ]
   {[declarations]*, [methods]*
    [interface method definition]1+ }
```

public class TextObject: DiagramObject, IScalable {...}
calling implemented methods

In the simplest case implemented methods can be called like other methods.

If the methods are invoked from a base class context that does not implement the methods than the object must first be tested to see if the interface is implemented and then cast.

```java
if (anObject is anInterface){
    anInterface thisInterface = (anInterface) anObject;
    thisInterface.aMethod(...);
}
```

The "as" operator will test for the implementation and cast if its implemented, otherwise it returns null.

```java
IScalable scalable = d as IScalable;
```

<< see IScalable1.cs example >>
Consider a diagram program composed of TextDiagram, VectorDiagram, and ImageDiagram objects that are subclasses of different classes. (eg, TextDiagram derives from Text – Text class defines editing behaviors). In the program you want to be able to scale a collection of these objects. << see IScalable2.cs example>>

```
Iscalable [] diagramObjects; // assume assignment

foreach (Iscalable scalable in diagramObjects) {
    scalable.scale();
}
```
multiple interfaces

C# support single inheritance and multiple implementation.

Interfaces can derive (subclass), and combine, other interfaces.

With multiple interfaces implementations there can be method name collisions, for example implementing methods from different vendors, or an implemented method and method defined in the implementing class.

Explicitly invoking an implemented method

```
...  
  InterfaceName1.methodName(...);
  IFoo.compress();
  ...
  InterfaceName2.methodName(...);
  IBar.compress();
```

With method name collision, all but one of the methods must be explicitly invoke.
Explicit implementation

Implicitly implemented methods can be abstract or virtual and overridden or new in derived classes.

The interface's implementation can be hidden from a client by using explicit implementation. With explicit implementation the method cannot be abstract or virtual.

```csharp
interface IInterface {
    int P {get; set;} } // P is a property

interface IDerived: IInterface {
    new int P(); } // P is a method

class aClass : IDerived {
    int IInterface.P { get { ... } } // explicit implement
    public int P() { ... } } // implicit
```

Property P above is hidden (shadowed) by method P.
Abstract Vs Interfaces

Abstract
"is - a" relationship exists
There can only be one
Design needs to have variable member fields.
Noun + verb

Interface
"can-be-done" relationship exists
“is – a” relationship also exists (w/o data or defined behaviors).
There can be many different ADTs with this capability
verb – interfaces don't have member fields
no versioning (new)

Given single inheritance, interface is probably the starting design point for any abstract DT (data type) your considering .

Ask: "Are the defining ADTs of this abstract DT defining a "capability of" or a "member-of" the DT?"
Arrays

Arrays type System.Array
Collection class with easy to use C-style syntax

element methods properties << see on-line doc>>
BinarySearch(), Clear(), Copy(), Sort(), Reverse(), Length, Rank, Initialize(), SetValue()

type[] arrayName;
...
arrayName = new type [size];
or
type[] arrayName = new type [size];
or
type[] arrayName = new type [size] { v1, v2, ..., v_size};
type[] arrayName = {v1, v2, ..., v_size};

array values set to default value of type.

First element is at index 0 and last element is at Length - 1.
multi-dimensions

Regular Multi dimensional arrays

```java
type[,] arrayName;
type[,] arrayName = new type [rows, columns];
type[,] arrayName = { {v1, v2,...vn}, ....,
    {vi, vi+1, ..., vi+n};
```

right most subscript moves the fastest (is nested)

Bounds checking is enforced.

```java
int [,] vector = new int[3,4];
vector[2,0] is valid, vector[1,4] is an error.
```

Jagged arrays  each dimension is a 1 dimension array

```java
type[][] arrayName;
type[][] arrayName = new type [rows][];
arrayName[0][i] = new type[sizei];
arrayName[1][i+1] = new type[sizei+1];
...
arrayName[k][l] = aValue; // assume type correctness
```

Note separate (C and C++ like) indexes "[]" with jagged arrays
Indexers

Indexers are properties that act like overloaded C++ [] operator

Within a class syntactically treat its contents as an array.

```csharp
public type this [ type arg ] {
    get { ... }
    set { ... }  }
... 
object[i] = aValue;     // using set and get
Console.WriteLine("object[{0}] = {1}", i, object[i]);
```

this refers to current object,
get and set must be defined.
<< see indexer.cs textbook example

and indexer2d.cs >>

The index values do not need to be integers, they can be defined wrt the application

<table>
<thead>
<tr>
<th>ListBoxTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>- strings: string[]</td>
</tr>
<tr>
<td>- ctr: int</td>
</tr>
<tr>
<td>+ ListBoxTest(string[])</td>
</tr>
<tr>
<td>+ Add(String): void</td>
</tr>
<tr>
<td>+ GetNumEntries(): int</td>
</tr>
<tr>
<td>+ indexer : string</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Generic Collection interfaces

**IEnumerator<T>** iterate / enumerate through collection using foreach statement

```csharp
public interface IEnumerable<T> : IEnumerable
```

**ICollection<T>** provides CopyTo(), and Count, ISReadOnly, ISSynchronized and SyncRoot properties

```csharp
public interface ICollection<T> : IEnumerable<T>, IEnumerable
```

**IList<T>** used by array-indexable dynamic collections

```csharp
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
```

**IDictionary<K,V>** used by key / value based collections

**IComparer<T>** compares two objects in collection for sorting
IEnumerable\<T\>

IEnumerable implementers must implement GetEnumerator();

```csharp
public class name : IEnumerable\<type\>
...

type collectionOfType;
...

generic IEnumerator\<type\> GetEnumerator() {
    foreach (type t in collectionOfType {
        yield return t; }
    }
}
```

**yield** keywork used only in IEnumerator\<T\> blocks

```csharp
yield return expression
```

*expression* has to be implicitly convertible to the yield type of the iterator.
IEnumerator

Implementing IEnumerator (non generic) in a class allows it to be processed with a foreach stmt

known types have non-generic IEnumerator implemented.

Often the collection class type will have a private class that implements the IEnumerator interface

encapsulate the implementation

close semantic association between collection and enumerator

Non-generic IEnumerator must implement 1 property and 2 methods

public object Current  returns current object (get only)
public bool MoveNext()  increment ndex check bounds
public void Reset()  set index to -1

public bool MoveNext() {
    index++;
    if (index >= object.Length)
        return false;
    else
        return true; }


The foreach stmt will implicitly invoke GetEnumerator (and the enumerator constructor) and in each iteration implicitly invoke MoveNext()
List<T> class

List<T> is a generic, dynamic, "array"

public class List<T> : IList<T>, ICollection<T>,
                   IEnumerateable<T>, IList, ICollection, IEnumberable

additional properties and methods << review on-line documentation >>

- Capacity get / set number of elements List can hold
- Count get / set number of elements in the List
- Item get / set element at index – this is an indexer for List (error in text – not a method)
- Add() adds object to list
- AddRange() adds elements of ICollection at end of list
- BinarySearch()...
- Clear() removes all elements
- Find()...
- GetEnumerator() returns enumerator for iteration
- Sort() sort List
IComparable<T>

Sort(), or, compare items in List<T> using IComparer<T>

IComparable<T> requires implementation of the CompareTo() method.

    int CompareTo ( T other )

Many .NET classes implements IComparable with
default case-sensitive implementation. (Int32, String ....)

    public sealed class String : IComparable, ICloneable,
    IConvertible, IComparable<string>, IEnumerable<string>,
    IEnumerable, IEquatable<string>

Comparing is implemented by calling the CompareTo()

    tObject.CompareTo(tObject2)

    returns:

    -1    object < object2
    0     object == object2
    1     object > object2
Collections

Non-generic collections

- **CollectionBase**: abstract class for strongly typed collections
- **DictionaryBase**: abstract class for key to value collections
- BitArray: array of boolean values (masks, sets)
- ArrayList: dynamic array
- Queue, Stack: FIFO and LIFO collections
- Hashtables: dictionary optimized for retrieval
- SortedList: dictionary sorted by keys

Collections implement the ICollections, IComparable interfaces.

More generic and non-generic collections used with GUI controls in Windows and Web Forms and other .NET libraries.
ArrayList

ArrayList is a non-generic dynamic "array".

```csharp
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
```

Not always sorted – use `Sort()`

additional properties and methods << review on-line documentation >>

<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>get / set number of elements arraylist can hold</td>
</tr>
<tr>
<td>Item(index)</td>
<td>get / set the element at index</td>
</tr>
<tr>
<td>Add()</td>
<td>add object to end of arraylist</td>
</tr>
<tr>
<td>Contains(object)</td>
<td>is element in arraylist</td>
</tr>
<tr>
<td>Insert(index, obj)</td>
<td>inserts element into arraylist</td>
</tr>
<tr>
<td>IsSynchronized</td>
<td>get bool is value thread safe? (abstract property)</td>
</tr>
<tr>
<td>Remove(obj)</td>
<td>removes first occurrence of specified object</td>
</tr>
<tr>
<td>RemoveAt(index)</td>
<td>removes element at index</td>
</tr>
<tr>
<td>ToArray()</td>
<td>convert to an array</td>
</tr>
<tr>
<td>TrimToSize()</td>
<td>sets capacity to actual number of elements</td>
</tr>
</tbody>
</table>
Queues, Stacks

Queues are FIFO collections

```csharp
public class Queue : ICollection, IEnumerable, ICloneable

properties and methods
    Enqueue(obj)       add object to end of queue
    Dequeue()          removes object at front of queue
    Peek()             returns object at front of queue w/o removal
```

Stacks are LIFO collections

```csharp
public class Stack : ICollection, IEnumerable, ICloneable

properties and methods
    Pop()               remove object at the top of the stack
    Push(obj)           insert object at the top of the stack
```
Hashtables

 Dictionary is a collection that associates a value with a key. The key is used for value access.

 Any object can be a key or a value.

 IDictionary provides Item property that returns the value with a specified key.

 Hashtables is a dictionary collection optimized for retrieval.

 Each value is stored in a bucket (numbered like offset to array).

 Each key must provide a int GetHashCode() method to convert key to a bucket address (hash function).

 Collisions occur when GetHashCode() returns the same bucket for more than one key.

 With bucket load factor of 1 (default) collisions are stored in an ordered list of values which is found w/ BinarySearch().