Interfaces

- Interfaces describe a group of related functionalities that can belong to any class or struct.
- Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types.
- An interface cannot contain fields or static members.
- Interface members are automatically public – you do not include any access modifier.
- Interface members are automatically abstract – you do not include the word abstract.
- When a class or struct is said to inherit an interface, it means that the class or struct provides an implementation for all of the members defined by the interface.
  - If a base class implements an interface, the derived class inherits that implementation.
- Classes and structs can inherit from interfaces in a manner similar to how classes can inherit a base class, with two exceptions:
  - A class or struct can inherit more than one interface.
  - When a class or struct inherits an interface, it inherits only the method names and signatures, because the interface itself contains no implementations (no method bodies).

Defining an Interface

- The declaration for an interface is similar to the declaration for a class – you use the ***interface*** keyword instead of class.
- Methods and properties that are declared within an interface can't include implementations – method declarations and get/set accessors always end with a semicolon.
- By convention, interface names begin with the letter **I** to distinguish them from classes.

General Form: Defining an Interface

```
public interface InterfaceName
{
    dataType PropertyName
    {
        get;
        set;   // don't include for a read-only property
    }

    returnType MethodName(parameters);
}
```

Example:

```
// any class that implements this interface will require
// a Length read-write property and a GetInfo( ) method

public interface IBoat
{
    int Length
    {
        get;
        set;
    }
    string GetInfo();
}
```

Interfaces Defined by the .NET Framework

- The .NET Framework defines many interfaces that you can implement in your classes…

Commonly Used .NET Interfaces

Interface	Members	Description
ICloneable	`object Clone()`	Creates a duplicate copy of an object.
IComparable	`int CompareTo(object)`	Compares the current instance with another object of the same type and returns an integer that indicates whether the current instance precedes, follows, or occurs in the same position in the sort order as the other object.

IConvertible	`decimal ToDecimal()`
	`int ToInt32()`
	`// and many others`
		Converts an object to one of the common language runtime types, such as Int32, Decimal, or Boolean.

IDisposable	`void Dispose()`	Frees unmanaged resources.

- Of these four, the one you're most likely to implement is ICloneable. This interface lets you create objects than can produce copies of themselves.

Commonly Used .NET Interfaces for Collections

Interface	Members	Description
IEnumerable	`IEnumerator GetEnemerator()`	Gets an enumerator for the collection.

IEnumerator	`object Current`
	`bool MoveNext()`
	`void Reset()`
		Defines an enumerator that provides read-only, forward-only access to a collection.

ICollection	`int Count`
	`bool IsSynchronized`
	`object SyncRoot`
	`void CopyTo(array, int)`
		Provides basic properties for an enumerable collection. This interface inherits IEnumerable.

IList	`[int] // an indexer`
	`int Add(object)`
	`void Clear()`
	`void Remove(object)`
	`void RemoveAt(int)`
		Manages a basic list of objects. This interface inherits ICollection and IEnumerable.

- IEnumerable and IEnumerator provide a standard mechanism for iterating through the items of a collection. If implemented, you can use the class in a foreach loop.

Generic Interfaces Defined by the .NET Framework

- The generic collection of the .NET Framework work with generic interfaces – like List<T>.
- Most of these generic interfaces have corresponding regular interfaces.

Commonly Used .NET Generic Interfaces

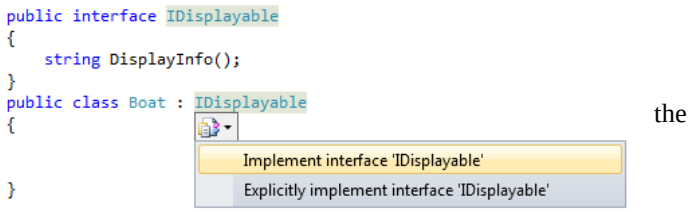
Interface	Members	Description
IComparable<>	`int CompareTo(T)`	Compares objects of type T
IEnumerable<>	`IEnumerator<T> GetEnemerator()`	Gets an enumerator of type T for the collection.

ICollection<>	`int Count`
	`bool IsReadOnly`
	`T SyncRoot`
	`void Add(T)`
	`void Clear()`
	`bool Contains(T)`
	`void CopyTo(array, int)`
	`bool Remove(T)`
		Provides basic properties for an enumerable collection. This interface inherits IEnumerable<>.

IList<>	`[int] // an indexer`
	`int IndexOf(T)`
	`void Insert(int, T)`
	`void RemoveAt(int)`
		Manages a basic list of objects. This interface inherits ICollection<> and IEnumerable<>.

How to Implement an Interface

- To declare a class that implements one or more interfaces, type a colon after the class name, then list the interfaces that the class implements.
- If a class inherits another class, you must include the name of the inherited class before the names of any interfaces the class implements.
- After you enter the name of an interface on a class declaration, you can automatically generate *stubs* for the members of the interface. Right-click on the name of the interface and select the Implement Interface command from the menu.




```
public interface IDisplayable
{
    string DisplayInfo();
}
public class Boat : IDisplayable
{
    public string DisplayInfo()
    {
        throw new NotImplementedException();
    }
}
```

The code stub generated for the implement IDisplayable interface…

- Replace the line of code that is automatically included in the body of the method with what you want the method to do.

Example: A Boat Class that Implements a User-Defined Interface and ICloneable

- The Boat class will explicitly inherit from Object to illustrate the order the items are placed in the class header…

```
// User-Defined Interface
public interface IDisplayable
{
    string DisplayInfo();
}

// Boat class that explicitly inherits from the Object class
// and implements user-defined IDisplayable and .NET's ICloneable
public class Boat : Object, IDisplayable, ICloneable
{
    private int length = 5;
    public int Length
    {
        get { return length; }
        set
        {
            if (value >= 5)
                length = value;
        }
    }
    // define DisplayInfo() from IDisplayable interface
    public string DisplayInfo()
    {
        return "Length: " + length;
    }
    // define Clone() from ICloneable interface
    public object Clone()
    {
        Boat boat = new Boat();
        boat.length = this.length;
        return boat;
    }
}
```

Interface Benefits

- Interfaces allow similar classes to have similar capabilities.
- The interface implementation relationship is a ***can do*** relationship: The class *can do* what the interface requires.
- Using interfaces allow for *polymorphism* – each class that implements the same interface will have the some of the same members but may implement them differently.
- Each class that implements the same interface can be treated as a data type of that interface.

Declarations:

```
public class Boat : IDisplayable
public class Person : IDisplayable

// each instance of Boat or Person can be treated as type IDisplayable…
IDisplayable obj1 = new Boat();
IDisplayable obj2 = new Person();

// both Boat and Person instances have the DisplayInfo() method
```

`obj1.DisplayInfo()` and `obj2.DisplayInfo()` both call unique implementations of the `DisplayInfo()` method.

Example

```
// User-Defined Interface
public interface IDisplayable
{
    string DisplayInfo();
}

// Person class that explicitly inherits from the Object class
// and implements user-defined IDisplayable interface

public class Person : Object, IDisplayable
{
    private string name = "Unknown Unknown";
    public string Name
    {
        get { return name; }
        set
        {
            // names must have 2 names - first and last
            if (value.Split().Length == 2)
                name = value;
        }
    }

    // define DisplayInfo() from IDisplayable interface
    public string DisplayInfo()
    {
        string[] names = name.Split(' ');
        if (names.Length == 1)
            return name;
        else
            return names[1] + ", " + names[0];
    }
}// end class
```

Test Code:

```
    private void Person_Click(object sender, EventArgs e)
    {
        Person newPerson = new Person();
        newPerson.Name = txtPersonName.Text;
        displayable.Add(newPerson);

        // call Display() which requires an object that implements IDisplayable
        Display(newPerson);

        string display = "Interface Example...\n";

        if (id is Person)
           display += id.DisplayInfo();

    }

    // any object implementing IDisplayable can use this method
    private void Display(IDisplayable obj)
    {
        txtTextbox.Text = obj.DisplayInfo();
    }
```