

# Module 10

## "Interfaces"



**TEKNOLOGISK**  
**INSTITUT**

# Agenda

- ▶ **Introducing Interfaces**
- ▶ Using Interfaces
- ▶ Building Comparable Objects with **IComparable**
- ▶ Building Enumerable Types with **IEnumerable**
- ▶ *Optional:* Using Types with **IDisposable**
- ▶ Lab 10
- ▶ Discussion and Review

# What is an Interface?

- ▶ An *interface* is a named set of abstract members

```
interface IDropTarget
{
    void OnDragDrop( DragEventArgs e );
    void OnDragEnter( DragEventArgs e );
    void OnDragLeave( EventArgs e );
    void OnDragOver( DragEventArgs e );
    bool AllowDrop { get; set; }
}
```

- ▶ It is a more or less rock-steady rule that interface names start with a capital **I**
- ▶ Interfaces can contain methods, properties, events declarations only
  - Cannot contain member variables, method bodies or implementation
- ▶ Interface methods are implicitly public, so access modifiers are disallowed
- ▶ An interface is a reference type

# Defining Custom Interfaces

- ▶ You can easily define your own interface types

```
interface IPointy
{
    int Points{ get; }
}
```

```
static void Main()
{
    IPointy p = new IPointy();
}
```

```
interface IPointy
{
    public int numberOfPoints;
    public IPointy()
    {
        numberOfPoints = 0;
    }
    int GetNumberOfPoints()
    {
        return numberOfPoints;
    }
}
```

- ▶ Interfaces does not really provide any substance until they're implemented by a concrete class or struct



# Interfaces vs Abstract Classes

## ► Differences

- Interfaces cannot contain implementation
- Abstract classes are used for partial implementation
- Interface members are all public
- Interfaces can derive only from other interfaces
- Interfaces are for types unrelated by inheritance – abstract classes enforce inheritance relationship

## ► Identical aspects

- Reference types
- Cannot be instantiated
- Not allowed to be sealed
- Can be derived from by classes

# Implementing an Interface

- ▶ The implementing method or property must be public and have the same signature as the interface method or property being implemented

```
public class Triangle : Shape, IPointy
{
    public Triangle( ) { }
    public override void Draw()
    {
        Console.WriteLine( "Drawing {0} the Triangle", PetName );
    }
    public int Points
    {
        get { return 3; }
    }
}
```

- ▶ Using Visual Studio eases interface implementation



# Agenda

- ▶ Introducing Interfaces
- ▶ **Using Interfaces**
- ▶ Building Comparable Objects with **IComparable**
- ▶ Building Enumerable Types with **IEnumerable**
- ▶ *Optional:* Using Types with **IDisposable**
- ▶ Lab 10
- ▶ Discussion and Review

# Invoking Members at the Object Level

- ▶ Invoke methods and properties directly

```
Triangle tri = new Triangle();  
Console.WriteLine("Points: {0}", tri.Points );
```

- ▶ Alternatively, you could explicitly convert to the interface type to check whether type implements the interface

```
Triangle tri = new Triangle();  
try  
{  
    IPointy pointy = (IPointy) tri;  
    Console.WriteLine( pointy.Points );  
}  
catch( InvalidCastException e )  
{  
    Console.WriteLine( e.Message );  
}
```



# The **is** and **as** Keywords for Interfaces

- ▶ If the object can be treated as implementing the interface, **as** returns a reference to such an interface

```
Triangle tri = new Triangle();  
IPointy pointy = tri as IPointy;  
if( pointy != null )  
{  
    Console.WriteLine( pointy.Points );  
}  
else { // Does not implement Ipointy }
```

- ▶ **is** can be used to check directly for implementation of a specific interface

```
if( tri is IPointy )  
{  
    Console.WriteLine( ( (IPointy) tri).Points );  
}  
else { // Does not implement Ipointy }
```



# Interfaces as Parameters and Return Values

- ▶ Interfaces are reference types and behave exactly like other reference types with respect to methods
- ▶ They can be passed to methods as parameters

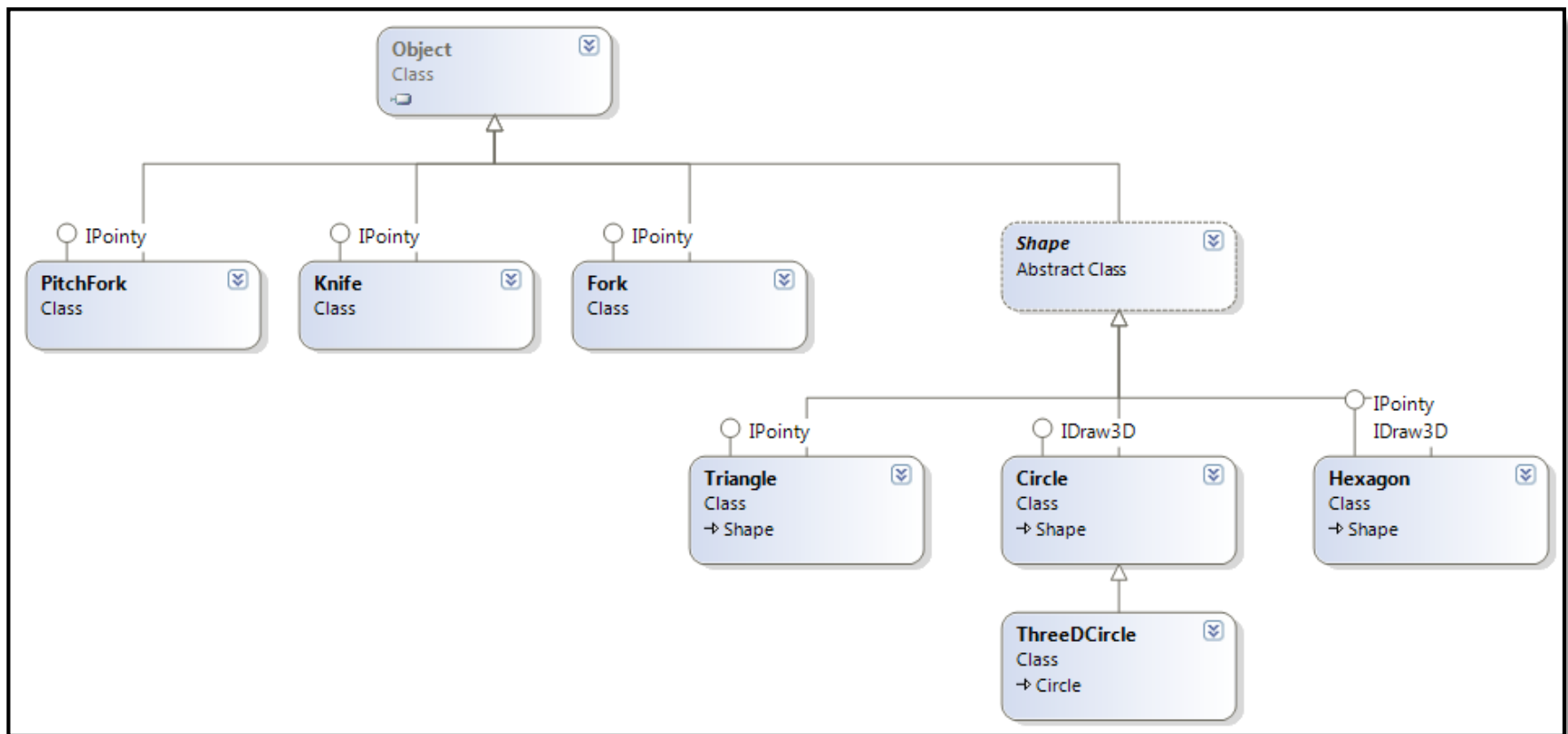
```
static void WritePointy( IPointy pointy )  
{  
    Console.WriteLine( pointy.Points );  
}
```

- ▶ Similarly, they can be returned from methods as return values

```
static IPointy ExtractPointyness( object o )  
{  
    return o as IPointy;  
}
```

# Arrays of Interface Types

- ▶ You can iterate through an array of interfaces and treat each item identically





# Multiple Inheritance with Interface Types

- ▶ A class can implement an arbitrary number of interfaces
  - But only have one superclass!

```
interface IDrawable
{
    void Draw();
}
```

```
interface IPrintable
{
    void Print();
    void Draw();
}
```

```
interface IRenderToMemory
{
    void Render();
}
```

```
class SuperShape : IDrawable, IPrintable, IRenderToMemory
{
    public void Draw() { ... }
    public void Print() { ... }
    public void Render() { ... }
}
```

- ▶ Potential name clash!



# Designing Interface Hierarchies

- ▶ An interface can extend an arbitrary number of interfaces
- ▶ Arrange your related interfaces into interface hierarchies!
- ▶ This has been done extensively through the .NET Framework classes
  - E.g. **IList**, **ICollection**, ...

```
interface IList : ICollection, IEnumerable
{
    ...
}
```

- ▶ An interface cannot be more accessible than it's base interface!



# Quiz: Designing Interfaces – Right or Wrong?

```
interface IDrawable  
{  
    void Draw();  
}
```

```
class WyattEarp : IDrawable  
{  
    void Draw() { ... }  
}
```

```
class Circle : IDrawable  
{  
    public void Draw() { Console.WriteLine("Drawing..."); }  
}
```

```
class Artist : IDrawable  
{  
    public void Draw( Canvas c ) { ... }  
}
```

# Agenda

- ▶ Introducing Interfaces
- ▶ Using Interfaces
- ▶ Quiz: Designing Interfaces
- ▶ **Building Comparable Objects with `Comparable`**
- ▶ Building Enumerable Types with `IEnumerable`
- ▶ *Optional*: Using Types with `IDisposable`
- ▶ Lab 10
- ▶ Discussion and Review

# The **IComparable** Interface

- Implement **IComparable** to compare objects to each other

```
interface IComparable
{
    int CompareTo( object obj );
}
```

CompareTo() Return Value	Indicating...
< 0	This instance is before <b>obj</b>
0	This instance is equal to <b>obj</b>
> 0	This instance is after <b>obj</b>

- Built into .NET



# Implementing **IComparable**

- ▶ You can implement **IComparable** in your own types

```
public class Car : IComparable
{
    public int ID { get; set; }
    ...

    public int CompareTo( object obj )
    {
        Car other = obj as Car;
        if( this.carID < other.carID ) { return -1; }
        else if( this.carID > other.carID ) { return 1; }
        return 0;
    }
}
```

```
Car c1 = ...;
Car c2 = ...;
if( c1.CompareTo( c2 ) < 0 )
{
    // c1 is less than c2
}
```

- ▶ **IComparable** types can be sorted e.g. in arrays



# The **IComparer** Interface

- ▶ Multiple sort orders can be obtained using the more general **IComparer**

```
interface IComparer
{
    int Compare( object o1, object o2 );
}
```

```
public class PetNameComparer : IComparer
{
    int IComparer.Compare( object o1, object o2 )
    {
        Car c1 = o1 as Car;
        Car c2 = o2 as Car;
        return String.Compare( c1.PetName, c2.PetName );
    }
}
```

```
Array.Sort( cars, new PetNameComparer() );
```



# Agenda

- ▶ Introducing Interfaces
- ▶ Using Interfaces
- ▶ Quiz: Designing Interfaces
- ▶ Building Comparable Objects with **Comparable**
- ▶ **Building Enumerable Types with IEnumerable**
- ▶ *Optional:* Using Types with **IDisposable**
- ▶ Lab 10
- ▶ Discussion and Review

# The **IEnumerable** Interface

- ▶ The **IEnumerable** interface states that the items of a class can be enumerated

```
using System.Collections;

interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
interface IEnumerator
{
    bool MoveNext ();
    object Current { get; }
    void Reset ();
}
```

- ▶ The **IEnumerator** interface provides an enumerator mechanism for the class
- ▶ Both are built into the .NET Framework base classes in the **System.Collections** namespace
- ▶ Arrays and collection types implement **IEnumerable** out-of-the-box



# Implementing **IEnumerable**

- ▶ You can implement **IEnumerable** in your own types

```
public class Garage : IEnumerable
```

```
{
```

```
    private Car[] carArray = new Car[ 4 ];
```

```
    public Garage()
```

```
    {
```

```
        carArray[ 0 ] = new Car()
```

```
        carArray[ 1 ] = new Car()
```

```
        carArray[ 2 ] = new Car()
```

```
        carArray[ 3 ] = new Car()
```

```
    }
```

```
    public IEnumerator GetEnumerator() { ... }
```

```
}
```

```
Garage garage = new Garage();  
foreach( Car c in garage )  
{  
    Console.WriteLine( c.PetName );  
}
```



# Agenda

- ▶ Introducing Interfaces
- ▶ Using Interfaces
- ▶ Quiz: Designing Interfaces
- ▶ Building Comparable Objects with **Comparable**
- ▶ Building Enumerable Types with **IEnumerable**
- ▶ *Optional: Using Types with **IDisposable***
- ▶ Lab 10
- ▶ Discussion and Review

# Objects, Values, and Scope

- ▶ Local variables live only throughout the scope in which they are declared
  - Fixed lifetime
  - Scheduled destruction
- ▶ Objects can outlive the scope in which they were allocated
  - Unbounded lifetime
  - Undetermined destruction
- ▶ Consequently; Objects are cleaned up by the *Garbage Collector*

```
static void Main()
{
    bool b = true;
    A longLivingVariable;
    if( b )
    {
        int i = 0;
        while( true )
        {
            A a = new A( i );
            if( ++i % 100 == 0 )
            {
                longLivingVariable = a;
            }
        }
    }
}
```



# Deallocating Objects

- ▶ There is no construct in C# to explicitly destroy objects
  - This is to avoid
    - Forgetting to destroy objects
    - Destroying more than once
    - Dangling references
    - ...
- ▶ The garbage collector *finalizes* the objects back into unused memory



# Defining Destructors

- ▶ Put cleanup logic in the destructor

```
class DataHandler
{
    FileStream fs;
    ...
    ~DataHandler()
    {
        fs.Close();
    }
}
```

- ▶ Similar to constructors, the destructor is named after the class (but with ~)
- ▶ Similar to constructors, destructors have no return type
- ▶ No access modifier is allowed
- ▶ Just a single destructor (with no parameters!) is allowed in each class



# Be Careful Out There!

- ▶ The finalization process takes place after “ordinary” garbage collection
- ▶ If your class has only managed resources, you should use a destructor!
- ▶ Avoid destructors whenever possible
  - Costs time
  - Hard to debug
  - Prolongs object life and memory usage
- ▶ Cannot know exactly when finalization takes place...!

# Disposing Classes

- ▶ Many .NET Framework classes implement **IDisposable**
  - You can also implement it yourselves
- ▶ You should always invoke **Dispose()** on objects if they implement **IDisposable**

```
using System.IO;
static void Main()
{
    FileStream fs =
        new FileStream( "myFile.txt", FileMode.OpenOrCreate );

    // These method calls do the same thing!
    fs.Close();    // WTF???
    fs.Dispose();
}
```

# The **using** Statement

- ▶ The **using** statement is a convenient shorthand to help you to remember to **Dispose()**

```
using( MyDisposableClass d = new MyDisposableClass() )  
{  
    d.DoStuff();  
    ...  
}
```

- ▶ **Dispose()** is always invoked at the end of the using block – even in the presence of exceptions!
- ▶ Strive to use **using** whenever possible instead of manually invoking **Dispose()**





# Lab 10: Working with Interfaces



# Discussion and Review

- ▶ Introducing Interfaces
- ▶ Using Interfaces
- ▶ Building Comparable Objects with **IComparable**
- ▶ Building Enumerable Types with **IEnumerable**
- ▶ *Optional*: Using Types with **IDisposable**



WINCUBATE

***Jesper Gulmann Henriksen***

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : [jgh@wincubate.net](mailto:jgh@wincubate.net)

WWW : <http://www.wincubate.net>

Hasselvangelen 243

8355 Solbjerg

Denmark