Interfaces

Overview

- Understanding interface-based programming
- Implementing an interface
- Obtaining interface references
- Interface hierarchies
- Multiple base interfaces

References

Interface-based Programming

The notion of interface-based programming exists already for years. In COM (ActiveX), the only way a client can communicate with a COM component is via an interface pointer (not a direct object reference).

From the conceptual standpoint, interfaces are contractual specifications between two pieces of code. That is, once an interface is defined and a class is defined as implementing that interface, clients of the class are guaranteed that the class has implemented all methods defined in the interface.
A Formal Definition

- An interface is nothing more than a collection of semantically related *abstract members*.

- The exact number of members defined by a given interface always depends on the exact behavior one is attempting to model.

- An interface specification is similar to a signature in algebra.
C# Syntax

// This interface defines the behavior of 'having points'.
public interface IPoints
{
    byte GetNumberOfPoints();
}

.NET interfaces are also able to support any number of properties and events.

// The points behavior as read/write property.
public interface IPoints
{
    byte Points { get; set; };
}

Implicitly virtual abstract.

Remove 'get' or 'set' to build read/write only property.
Pure Protocol

- Interfaces *never* define data types.

- Interfaces *never* provide a default implementation of methods.

- Every member of an interface is automatically abstract (methods are virtual abstract).

- C# only supports single inheritance, the interface-based protocol allows a given type to support numerous behaviors, while avoiding the issues that arise when deriving from multiple base classes (as in C++).
Implementing an Interface

When a C# class (or structure) chooses to extend its functionality by supporting a given interface, it does so using a comma-delimited list in the class definition.

```csharp
public class Hexagon : Shape, IPoints
{
    public Hexagon() { ... }
    public Hexagon( string aName ) : base( aName ) { ... }

    // inherited method
    public override void Draw() { ... }

    // IPoints Implementation
    public byte GetNumberOfPoints() { return 6; }
}
```

Direct base class must be listed first!
A Shape Hierarchy

Object

Shape

Hexagon

Circle

Triangle

“COM lollipop” notation

IPoints

IPoints

Com S 430
Obtaining an Interface

- C# provides a number of ways to obtain an interface reference from a type.

First approach: explicit cast

```csharp
Hexagon hex = new Hexagon( "Edward, The Black Prince" );
IPoints Ilptr = (IPoints)hex;
System.Console.WriteLine( Ilptr.GetNumberOfPoints() );
```

If one attempts to access an interface not supported by a given class using a direct cast, the system throws an InvalidCastException.
Circle c = new Circle("William I, the Conqueror");
IPoints ll ptr;

try
{
    ll ptr = (IPoints)c;
    System.Console.WriteLine(ll ptr.GetNumberOfPoints());
}
catch(InvalidCastException e)
{
    System.Console.WriteLine("OOPS! Not IPoints …");
}
Using “as”

Hexagon hex = new Hexagon( “Richard, Duke of Gloucester” );
IPoints lIptr;

lIptr = hex as IPoints;

if ( lIptr != null )
    System.Console.WriteLine( lIptr.GetNumberOfPoints() );
else
    System.Console.WriteLine( “OOPS! Not IPoints …” );

The “as” syntax sets the interface variable to null if a given interface is not supported by the object.
Using “is”

Hexagon hex = new Hexagon( "Edward I, Longshanks" );

// Are you IPoints?
if ( hex is IPoints )
    System.Console.WriteLine( hex.GetNumberOfPoints() );
else
    System.Console.WriteLine( "OOPS! Not IPoints …" );
Name Clashes

Problem:

- The base class and an interface both define the same method.
- A class implements several interfaces, but these interfaces define the same method.

```java
// A 3D drawing behavior
public interface IDraw3D {
    void Draw();
}
```
public class Line : Shape, IDraw3D
{
    // inherited method; interface method
    public override void Draw()
    {
        System.Console.WriteLine("Drawing a line …");
    }
}

// Calls Line.Draw()
Line l = new Line();
l.Draw();

// Also calls Line.Draw()
IDraw3D llptr = (IDraw3D)l;
llptr.Draw();
Identical Interfaces

// Three interfaces each defining identical methods
public interface IDraw
{
    void Draw();
}

public interface IDraw3D
{
    void Draw();
}

public interface IDrawToPrinter
{
    void Draw();
}
Explicit Interface Implementation

public class SuperImage : IDraw, IDraw3D, IDrawToPointer
{
    public void IDraw.Draw() { // Basic drawing logic }

    public void IDraw3D.Draw() { // 3D support }

    public void IDrawToPrinter.Draw() { // Printer logic }
}
// The base interface
public interface IDraw
{
    void Draw();
}

public interface IDraw2 : IDraw
{
    void DrawToPrinter();
}

public interface IDraw3 : IDraw2
{
    void DrawToMetaFile();
}
Using Interface Hierarchies

SuperImage si = new SuperImage();

// Get IDraw
IDraw ll ptrl Draw = (IDraw)si;
ll ptrl Draw.Draw();

// Now get IDraw3
if (ll ptrl Draw is IDraw3)
{
    IDraw3 ll ptrl Draw3 = (IDraw3)ll ptrl Draw;
    ll ptrl Draw3.DrawToMetaFile();
    ll ptrl Draw3.DrawToPrinter();
}
Multiple Base Interfaces

In C# it is completely permissible to create an interface that derives from multiple base interfaces (unlike COM).

```csharp
public interface IBasicVehicle
{
    void Drive();
}

public interface IUnderwaterVehicle
{
    void Dive();
}

public interface I007Car : IBasicVehicle, IUnderwaterVehicle
{
    void LaunchRockets();
}
```