Interfaces

CSIS 3540

Client Server Systems

Class 03

Topics

- Interfaces overview
- Defining and implementing interfaces
- Invoking interfaces
- Interfaces as parameters and return values
- Arrays of interface types
- Designing interface hierarchies
- .NET interfaces
  - IEnumerable, IEnumerator
  - ICloneable
  - IComarable, IComparer

Interface

- An interface is a class/type with only abstract members – no implementation!
- Abstract base classes have members that ONLY can be implemented by derived class
- Interfaces, however, can be implemented by ANY class.
  - Classes can implement multiple interfaces
  - Any object that implements an interface can have its methods invoked consistently independent of the object type!

```
public abstract class Parent {
   abstract int myMethod();
}

public class Child : Parent {
   override public int myMethod { return 3; }
}

public interface ParentInterface {
   int myMethod2();  // interface methods are public and abstract
   int myMethod3();
}

public class Kid : ParentInterface { … }  // implement methods 2 and 3

public class Child : Parent, ParentInterface { … }  // implement all three methods
```

Classes and Interfaces

- Interfaces are HIGHLY polymorphic
  - Multiple Inheritance!
  - Interface x : y, z, q
  - Interface m : x, p, r
- Classes are NOT polymorphic
  - Children have one parent
  - Class x : y
  - Class z : y
  - Class q : x

ICloneable – see ICloneableExample

- makes a copy of the object
- derived class must implement the Clone() method
  - can determine type of Clone() – deep, shallow, mixed
- Each object has a unique hash code (see use of GetHashCode method below)
  - Allows us to determine what is a clone vs original

```
 static void Main(string[] args)
    {
        Console.WriteLine("***** A First Look at Interfaces *****\n");

        // string class supports the ICloneable interface.
        string myStr = "Hello";

        // Get info about a unix operating system, implements ICloneable interface

        OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());

        // set up a new connection to a SQL database
        // SQLConnection implements ICloneable interface
        System.Data.SqlClient.SqlConnection sqlCnn =
            new System.Data.SqlClient.SqlConnection();

        // all of the above objects can be passed into method taking ICloneable.
        CloneMe(myStr);
        CloneMe(unixOS);
        CloneMe(sqlCnn);

        // now clone unixOS and display
        OperatingSystem osClone = ReturnAClone(unixOS) as OperatingSystem;
        Console.WriteLine($"osClone: {osClone.GetType().Name} {osClone.GetHashCode()}" +
            $", original is {unixOS.GetType().Name} {unixOS.GetHashCode()}");
        Console.ReadLine();
    }

    /// <summary>
    /// Clone an object that implements ICloneable and display information about the cloned object
    /// </summary>
    /// <param name="c">object to be cloned</param>
    private static void CloneMe(ICloneable c)
    {
        // Clone whatever we get and print out the name.
        object theClone = c.Clone();
        Console.WriteLine($"Your clone is a: {theClone.GetType().Name} {theClone.GetHashCode()}, " +
            $"original is {c.GetType().Name} {c.GetHashCode()}");
    }

    private static ICloneable ReturnAClone(ICloneable c)
    {
        return (ICloneable)c.Clone();
    }
}
```

CloneablePoint

- Our venerable Point object can be cloned as it implements IClonable

```
Console.WriteLine("***** Fun with Object Cloning *****\n");
Console.WriteLine("Cloned startingPoint and stored new Point in clonedPoint");
Point startingPoint = new Point(100, 100, "Jane");
Point clonedPoint = (Point)startingPoint.Clone();

Console.WriteLine("Before modification:");
Console.WriteLine($"startingPoint: {startingPoint}");
Console.WriteLine($"clonedPoint: {clonedPoint}");

// now change clonedPoint

clonedPoint.desc.PointName = "My new Point";
clonedPoint.X = 9;

Console.WriteLine("\nChanged clonedPoint.desc.PointName and clonedPoint.X");
Console.WriteLine("After modification:");
Console.WriteLine($"startingPoint: {startingPoint}");
Console.WriteLine($"clonedPoint: {clonedPoint}");
```

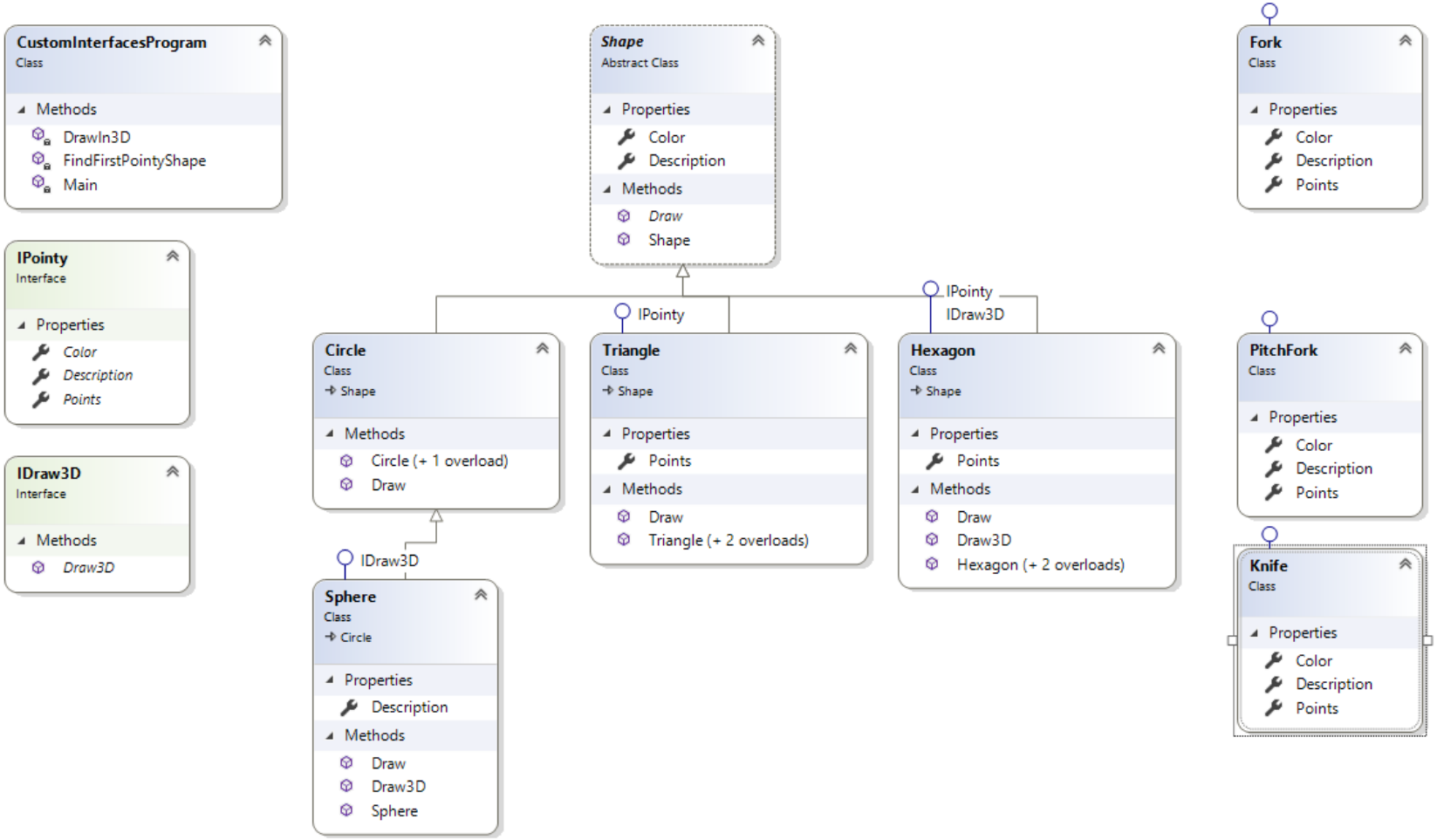
```
public class Point : ICloneable
{
    --------- missing code here, see original source code

    // Return a copy of the current object.
    // Now we need to adjust for the PointDescription member.
    public object Clone()
    {
        // First get a shallow copy.
        Point newPoint = (Point)this.MemberwiseClone();

        // Then fill in the gaps.
        // COMMENT THIS OUT TO SHOW SHALLOW COPY EXAMPLE

        PointDescription currentDesc = new PointDescription
        {
            PointName = this.desc.PointName
        };
        newPoint.desc = currentDesc;
        return newPoint;
    }

}
```

Custom Interfaces

- See
  - CustomInterfaces
- Study this together
- Note that properties in interfaces are OK!! They are actually methods (in disguise).
  - **But not fields.**

IPointy Class Diagram

- Notice circle notation indicating implementing an interface



Casting and Interfaces

- Use as
  - IPointy thing = otherThing as IPointy;
  - instead of a cast
    - IPointy thing = (IPointy) otherThing;
  - Reason?
    - If otherThing does NOT implement IPointy, returns null to thing
    - If you use a cast, an exception is thrown.
- Use is
  - if(thing is IPointy) thing.myColor = ConsoleColor.Black;

Interfaces as parameters

- Any object that implements an interface can be passed as a parameter of the interface type.
  - One could then inspect the object passed and take some appropriate action

```
    shapes[i].Draw();

    // Who's pointy?
    if (shapes[i] is IPointy)
    {
        IPointy ip = shapes[i] as IPointy;
        Console.WriteLine("-> Points: {0} Color: {1}", ip.Points, ip.Color);
    }
    else
        // If this is a sphere, the description will not display!
        Console.WriteLine("-> {0} is not pointy!", shapes[i].Description);
    Console.WriteLine();

    // Can I draw you in 3D?
    if (shapes[i] is IDraw3D)
        DrawIn3D(shapes[i] as IDraw3D);
```

Interfaces as return values

- Interfaces can also be used as method return values.
  - an array of Shape objects and returns a reference to the first item that supports IPointy.

```
// This method returns the first object in the
// array that implements IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
    foreach (Shape s in shapes)
    {
        if (s is IPointy)
            return s as IPointy;
    }
    return null;
}
```

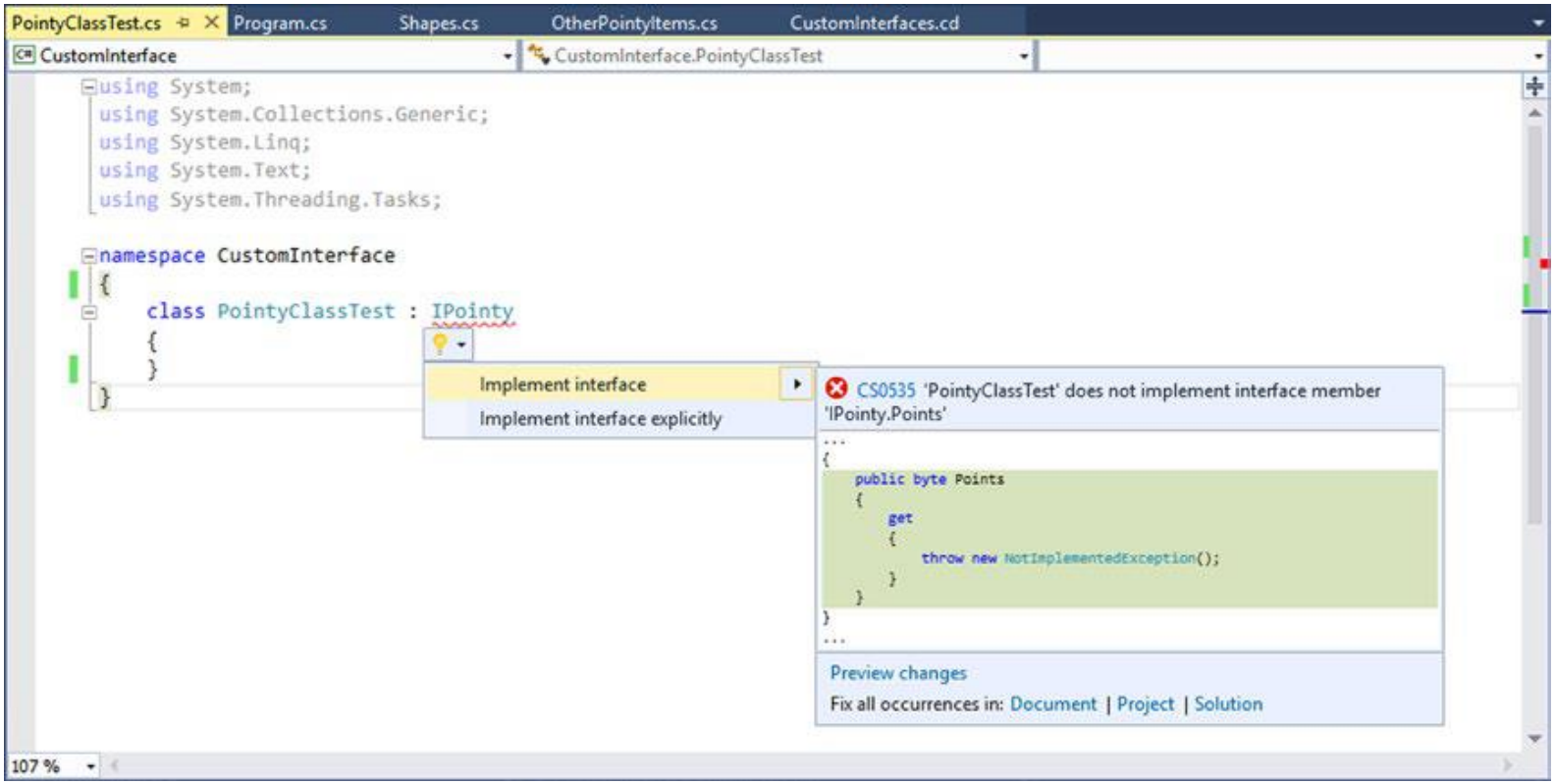
Arrays of interface

- You can create an array of an interface type consisting of objects that implement that interface.

```
// Make an array of Shapes.
Shape[] shapes = {
    new Hexagon("Hexy", ConsoleColor.Red),
    new Circle(),
    new Sphere("Ball"),
    new Triangle("Joe", ConsoleColor.Cyan),
    new Circle("JoJo"),
    // new Knife() -- Won't compile, is not a Shape!
};

IPointy[] pointyObjects = {
    new Hexagon("Hexy", ConsoleColor.Red),
    new Triangle("Joe", ConsoleColor.Cyan),
    new PitchFork(),
    new Knife()
    // new Sphere("Earth") -- Won't compile! Is not Pointy!
};
```

Implementing Interfaces in VS

- Type in interface when extending class, and VS will give you the option of implementing interface members
- Will include an exception to start
- Can be made explicit (fully qualified name)



Hierarchies and Multiple Inheritance

- Interfaces can extend interfaces (hierarchy)
  - See InterfaceHierarchy project
  - public interface IAdvancedDraw : IDrawable
- Interfaces can extend MULTIPLE base interfaces
  - See MultipleInterfaceHierarchy
  - If there is a name clash, either implement one method, or qualify the name and implement all.
    - See InterfaceNameClash

IEnumerable, IEnumerator - Iteration

- Allows working with foreach (Iteration)
- IEnumerable – aggregator (like an array) implements this
  - GetEnumerator() needs to be implemented
- IEnumerator – provides the mechanisms for iteration
  - MoveNext(), Current(), Reset()
  - Arrays and Lists already implement this
- See **CustomEnumerator** and **CustomEnumeratorWithYield**

```
public class Garage : IEnumerable
{
    private Car[] myAutos;

    /// <summary>
    /// Fill with some Car objects upon startup.
    /// </summary>
    public Garage()
    {
        myAutos = new Car[]
        {
            new Car() { CarName = "Rusty", CurrentSpeed = 80, CarID = 1},
            new Car() { CarName = "Mary", CurrentSpeed = 40, CarID = 234},
            new Car() { CarName = "Viper", CurrentSpeed = 40, CarID = 34},
            new Car() { CarName = "Mel", CurrentSpeed = 40, CarID = 4},
            new Car() { CarName = "Chucky", CurrentSpeed = 40, CarID = 5},
            // null
        };
    }

    /// <summary>
    /// Return the enumerator for the array of cars.
    /// If you comment out IEnumerable and the code below, foreach will not compile!
    /// It requires GetEnumerator.
    /// </summary>
    /// <returns></returns>

    public IEnumerator GetEnumerator()
    {
        // Return the array object's IEnumerator.
        return myAutos.GetEnumerator();
    }
}
```

IComparable and IComparer

- IComparable – the class is “comparable” - implemented by Array
- IComparer is used by Array.Sort()
  - Method: Compare(object a, object b) which returns -1,0,1 for less than, equal, or greater then
- See **ComparableCar**: look at CommonCar.cs
- **SportsLeague (lab due today)** example below – hint extra credit!!

```
        Array.Sort(athletes, 0, teamSize, new AthleteCompare()); // sort the array

        for(int i = 0; i < teamSize; i++)
        {
            Athlete a = athletes[i];
            Console.WriteLine("{0} {1} {2} (age: {3})", a.position, a.firstName, a.lastName, a.age);
        }

        Console.WriteLine();

    }

    // comparator used to sort the array by last names

    public class AthleteCompare : IComparer<Athlete>
    {
        public int Compare(Athlete a, Athlete b)
        {
            return a.lastName.CompareTo(b.lastName);
        }
    }
```