ІНТЕРФЕЙСИ

Питання 5.2.

Поняття інтерфейсних типів

- Інтерфейс являє собою просто іменований набір абстрактних членів.
  - абстрактні методи є чистим протоколом, оскільки вони не надають стандартної реалізації.
  - Специфічні члени, які визначаються інтерфейсом, залежать від того, яке точно поведінку він моделює.
  - Іншими словами, інтерфейс висловлює поведінку, яку задані клас або структура можуть обрати для підтримки.
- Абстрактний клас також може визначати будь-яку кількість конструкторів, полів даних, неабстрактних членів (з реалізацією) і т.п.
  - Інтерфейси можуть містити тільки визначення абстрактних членів (до C# 8.0).
- Поліморфний інтерфейс, встановлений абстрактним батьківським класом, володіє серйозним обмеженням: визначені ним члени підтримуються ***тільки породженими типами***.
  - Проте в крупних програмних системах дуже чатсо розробляються численні ієрархії класів, які не мають спільного батька, за винятком System.Object.
  - Для абстрактного класу не існує способу налаштування типів у різних ієрархіях для підтримки одного поліморфного інтерфейса.

Приклад

```
public abstract class CloneableType
{
  // Только производные типы могут поддерживать этот
  // "полиморфный интерфейс". Классы в других иерархиях
  // не имеют доступа к этому абстрактному члену.
  public abstract object Clone();
}
```

- При такому визначенні підтримувати метод Clone() можуть тільки члени, які розширяють клас CloneableType.
  - Якщо створюється новий набір класів, що не розширюють цей базовий клас, скористатися поліморфним інтерфейсом не вдасться.
  - Крім того, C # не підтримує множинне наслідування для класів:

```
// Нельзя! Множественное наследование для классов в C# не разрешено
public class MiniVan : Car, CloneableType
{
}
```

Допомагають інтерфейсні типи

- Після того, як інтерфейс визначено, він може бути реалізований будь-яким класом або структурою, в будь-якій ієрархії і всередині будь-яких простору імен або збірки.
  - Візьмемо стандартний .NET-інтерфейс Icloneable з простору імен.
  - Він визначає один метод Clone():

```
public interface ICloneable
{
  object Clone();
}
```

  - Інтерфейс реалізується багатьма непов'язаними типами (System.Array, System.Data.SqlClient.SqlConnection, System.OperatingSystem, System.String та ін).
  - Хоч ці типи не мають спільного батька (крім System.Object), їх можна трактувати поліморфним чином через інтерфейсний тип ICloneable.
  - Наприклад, якщо є метод CloneMe(), який приймає параметр інтерфейсного типу ICloneable, *цьому методу можна передавати будь-який об'єкт, який реалізує вказаний інтерфейс.*

Приклад та результати виводу

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** A First Look at Interfaces *****\n");
    // Все эти классы поддерживают интерфейс ICloneable.
    string myStr = "Hello";
    OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());
    System.Data.SqlClient.SqlConnection sqlCnn =
      new System.Data.SqlClient.SqlConnection();
    // Следовательно, все они могут быть переданы методу, принимающему ICloneable.
    CloneMe(myStr);
    CloneMe(unixOS);
    CloneMe(sqlCnn);
    Console.ReadLine();
  }
  private static void CloneMe(ICloneable c)
  {
    // Клонировать то, что получено, и вывести его имя.
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}",
      theClone.GetType().Name);
  }
}
```

```
***** A First Look at Interfaces *****

Your clone is a: String
Your clone is a: OperatingSystem
Your clone is a: SqlConnection
```

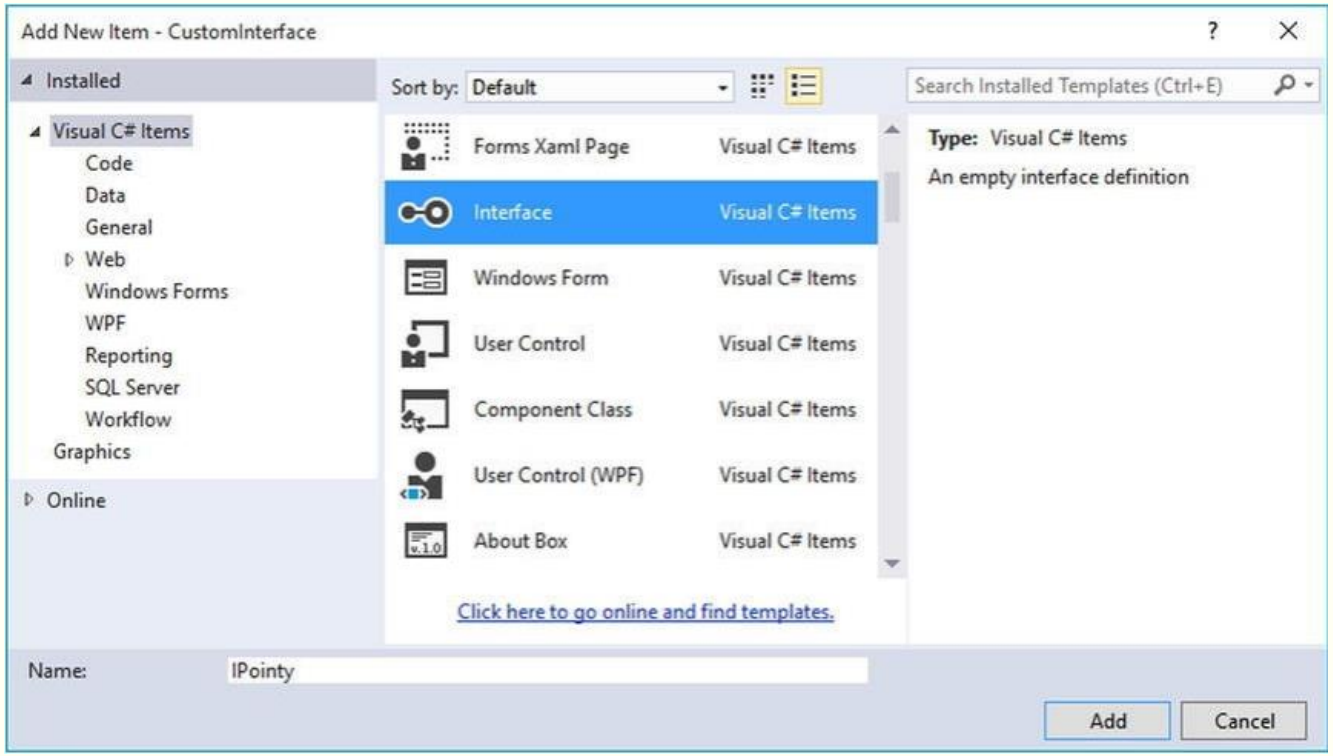
Інше обмеження абстрактних базових класів

- ***Кожний породжений тип повинен надати реалізації*** для всього набору абстрактних членів.
  - Для прикладу з фігурами нехай у базовому класі Shape визначено новий абстрактний метод GetNumberOfPoints(), який дозволяє породженим типам повертати кількість вершин, потрібних для візуалізації фігури:

```
abstract class Shape
{
  ...
  // Каждый производный класс должен теперь поддерживать этот метод!
  public abstract byte GetNumberOfPoints();
}
```

  - Очевидно, що спочатку єдиним класом, який в принципі має вершини, є Hexagon.
  - Проте через внесене оновлення кожний породжений клас (Circle, Hexagon і ThreeDCircle) повинен надати конкретну реалізацію GetNumberOfPoints(), навіть якщо це беззмістовно.

Визначення спеціальних інтерфейсів



- На синтаксичному рівні будь-який інтерфейс визначається за допомогою ключового слова interface мови C#.
  - На відміну від класів, для інтерфейсів ніколи не вказується базовий клас (навіть System.Object; хоч можуть задаватись базові інтерфейси).
  - Крім того, для членів інтерфейсу ніколи не вказуються модифікатори доступу (всі члени інтерфейсу є неявно відкритими і абстрактними).
  - Інтерфейси – це чистий протокол, тому реалізація для них ніколи не надається (проте у C# 8.0 введено поняття інтерфейсних методів за умовчанням).

```
// Этот интерфейс определяет поведение "наличия вершин"
public interface IPointy
{
  // Член является неявно открытым и абстрактным.
  byte GetNumberOfPoints();
}
```

Наступна версія IPointy призведе до видачі різних помилок компіляції

```
// Внимание! В этом коде полно ошибок!
public interface IPointy
{
  // Ошибка! Интерфейсы не могут иметь поля данных!
  public int numbOfPoints;

  // Ошибка! Интерфейсы не могут иметь конструкторы!
  public IPointy() { numbOfPoints = 0;};

  // Ошибка! Интерфейсы не могут предоставлять реализацию членов!
  byte GetNumberOfPoints() { return numbOfPoints; }
}
```

- Проте інтерфейсні типи .NET можуть також визначати будь-яка кількість прототипів властивостей.

Властивості в інтерфейсах

```
// Определение свойства, доступного только для чтения
public interface IPointy
{
  // Свойство, доступное для чтения и для записи,
  // в этом интерфейсе может выглядеть так:
  // retVal PropName { get; set; }
  //
  // а свойство, доступное только для записи - так:
  // retVal PropName { set; }

  byte Points { get; }
}
```

- Інтерфейси нічого особливого не дають до тих пір, поки не будуть реалізовані класом або структурою.
  - Наприклад, розміщувати типи інтерфейсів таким же чином, як класи або структури, неможливо.

```
// Внимание! Размещать типы интерфейсов не допускается!
static void Main(string[] args)
{
  IPointy p = new IPointy();  // Ошибка на этапе компиляции!
}
```

- Інтерфейсні типи також можуть містити визначення подій і індексаторів.
  - Тут IPointy є інтерфейс, який виражає поведінку "наявності вершин".
  - Ідея проста: деякі класи в ієрархії фігур (наприклад, Hexagon) мають вершини, а деякі (як Circle) - ні.

```
// Этот класс порожден от System.Object
// и реализует единственный интерфейс.
public class Pencil : IPointy
{...}

// Этот класс тоже порожден от System.Object
// и реализует единственный интерфейс.
public class SwitchBlade : object, IPointy
{...}

// Этот класс порожден от специального базового
// класса и реализует единственный интерфейс.
public class Fork : Utensil, IPointy
{...}
```

```
// Эта структура неявно порождена от System.ValueType
// и реализует два интерфейса.
public struct PitchFork : IClonable, IPointy
{...}
```

```
// Новый производный от Shape класс по имени Triangle.
class Triangle : Shape, IPointy
{
  public Triangle() { }
  public Triangle(string name) : base(name) { }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Triangle", PetName); }

  // Реализация IPointy.
  public byte Points
  {
    get { return 3; }
  }
}
```

Реалізація інтерфейсу

- Щоб розширити функціональність класу (або структури) за рахунок підтримки інтерфейсів, необхідно додати в його визначення список потрібних інтерфейсів, розділених комами.
  - Прямий базовий клас повинен бути першим у цьому списку, тобто відразу ж після двокрапки.
  - Коли тип класу породжений безпосередньо від System.Object, допускається перераховувати тільки підтримувані інтерфейси, тому що компілятор C# автоматично розширює типи від System.Object, якщо не вказано інакше.

Реалізація інтерфейсу працює за принципом "все або нічого".

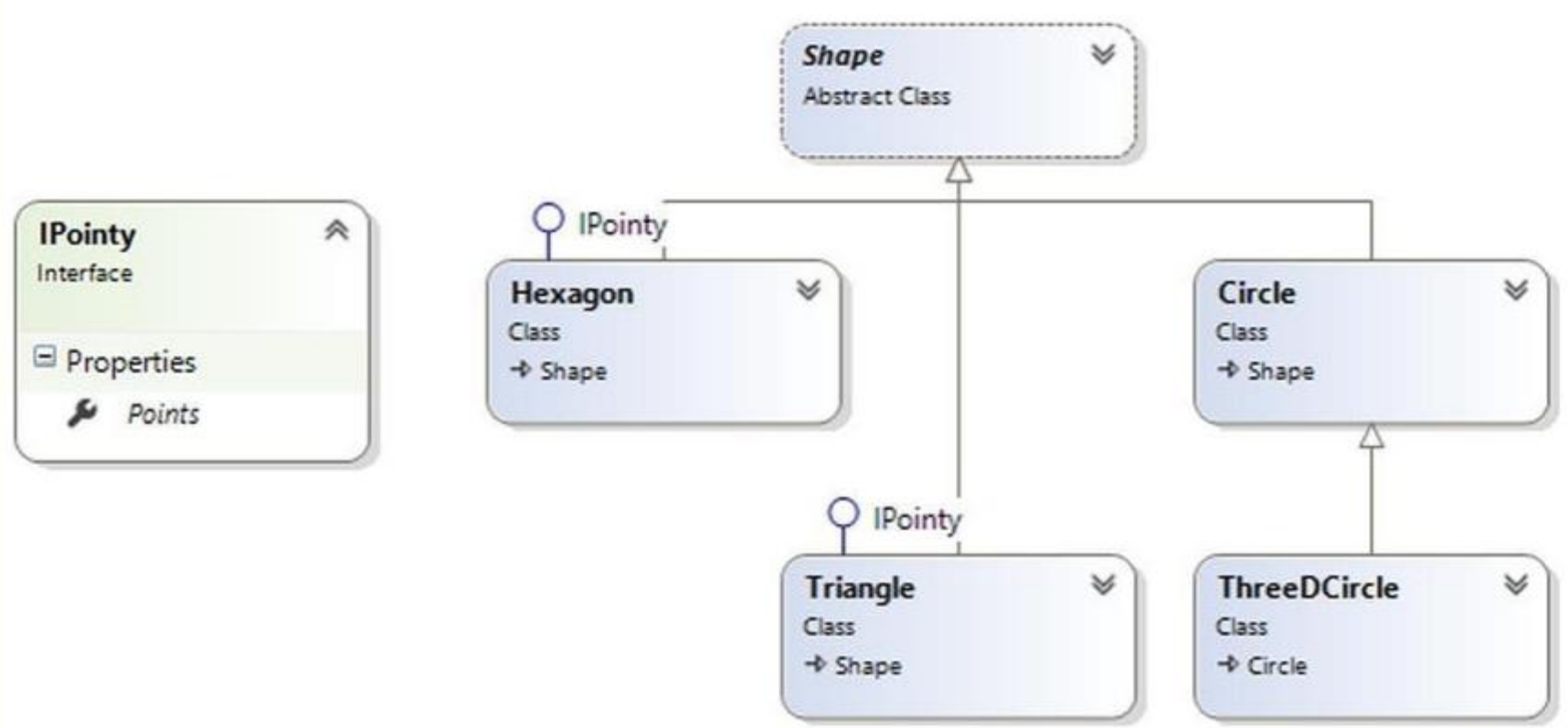
- Підтримуючий тип не має можливості вибирати, які члени повинні бути реалізовані, а які - ні.

Змінимо існуючий тип Hexagon так, щоб він теж підтримував інтерфейс IPointy

- Враховуючи, що в інтерфейсі IPointy визначено єдину доступну тільки для читання властивість, накладні витрати невеликі.
  - Однак у разі реалізації інтерфейсу, який визначає десять членів, тип буде відповідати за надання деталей для всіх десяти абстрактних членів.

```
// Hexagon now implements IPointy.
class Hexagon : Shape, IPointy
{
  public Hexagon(){ }
  public Hexagon(string name) : base(name){ }
  public override void Draw()
  { Console.WriteLine("Drawing {0}  the Hexagon", PetName); }
  // Реализация IPointy.
  public byte Points
  {
    get { return 6; }
  }
}
```

В цілому



- Зверніть увагу, що Circle і ThreeDCircle не реалізують IPointy, оскільки ця поведінка не має сенсу в даних класах.

Вызов членов интерфейса на уровне объектов

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  // Вызвать свойство Points, определенное интерфейсом IPointy.
  Hexagon hex = new Hexagon();
  Console.WriteLine("Points: {0}", hex.Points);
  Console.ReadLine();
}
```

- Такий підхід нормально працює в даному випадку, оскільки тут точно відомо, що тип Hexagon реалізує згаданий інтерфейс, а тому підтримує властивітьс Points.
  - Проте в інших випадках визначити, які інтерфейси підтримують даний тип, може бути неможливо.
  - Наприклад, нехай є масив з 50 об'єктів Shape-сумісних типів, причому тільки деякі з них підтримують IPointy.
  - Очевидно, що при спробі звернення до властивості Points для типу, який не реалізує IPointy, виникне помилка.
  - Як динамічно визначити, чи підтримує клас або структура потрібний інтерфейс?

```
static void Main(string[] args)
{
  ...
  // Перехватить возможное исключение InvalidCastException.
  Circle c = new Circle("Lisa");
  IPointy itfPt = null;
  try
  {
    itfPt = (IPointy)c;
    Console.WriteLine(itfPt.Points);
  }
  catch (InvalidCastException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

```
static void Main(string[] args)
{
  ...
  // Может ли hex2 интерпретироваться как IPointy?
  Hexagon hex2 = new Hexagon("Peter");
  IPointy itfPt2 = hex2 as IPointy;
  if(itfPt2 != null)
    Console.WriteLine("Points: {0}", itfPt2.Points);
  else
    Console.WriteLine("OOPS! Not pointy...");
  Console.ReadLine();
}
```

- ***(Варіант 1) Застосування явного зведення.***
  - Якщо тип не підтримує потрібний інтерфейс, згенерується виняток InvalidCastException.
  - Акуратно обробимо такі винятки.
  - в идеале хотелось бы выяснять, какие интерфейсы поддерживаются, перед обращением к их членам.
- ***(Варіант 2) Отримання посилань на інтерфейси за допомогою ключового слова as.***
  - Якщо тип об’єкту може бути інтерпретований як зазначений інтерфейс, то повертається посилання на цей інтерфейс, а якщо ні, то null-посилання.
  - Відпадає необхідність у логіці try / catch, оскільки не-null-посилання вказує на використання дійсного посилання на інтерфейс.

Отримання посилань на інтерфейси: ключове слово is

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  // Создать массив элементов Shape.
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")} ;
  for(int i = 0; i < myShapes.Length; i++)
  {
    // Вспомните, что базовый класс Shape определяет абстрактный
    // член Draw(), поэтому все фигуры знают, как себя рисовать.

    myShapes[i].Draw();
    // У каких фигур есть вершины?
    if(myShapes[i] is IPointy)
      Console.WriteLine("-> Points: {0}", ((IPointy) myShapes[i]).Points);
    else
      Console.WriteLine("-> {0}\'s not pointy!", myShapes[i].PetName);
    Console.WriteLine();
  }
  Console.ReadLine();
}
```

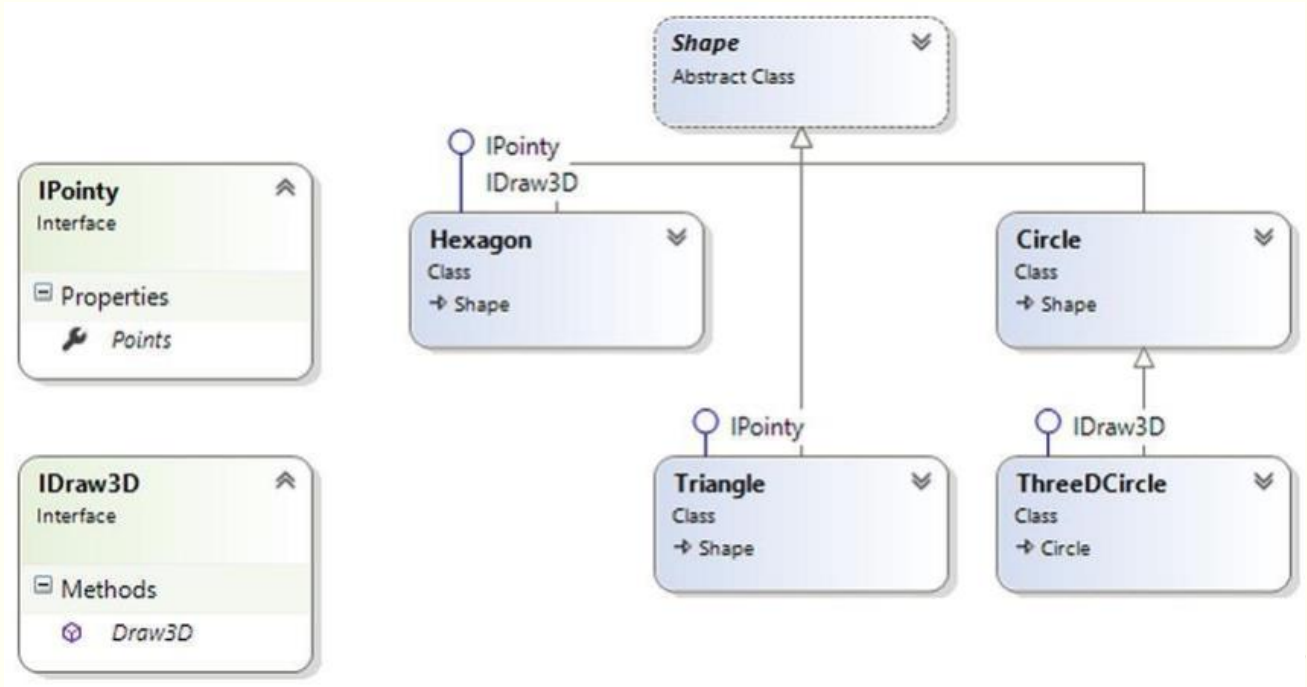
- Якщо об’єкт не сумісний із вказаним інтерфейсом, повертається значення false.
  - При сумісності можна безпечно звертатись до його членів без застосування логіки try/catch.
- Вивід програми:

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

Використання інтерфейсів у якості параметрів

- Оскільки інтерфейси – допустимі типи .NET, можна будувати методи, які приймають інтерфейси в якості параметрів.

```
// Моделирует способность визуализировать тип в трехмерном виде
public interface IDraw3D
{
  void Draw3D();
}
```




```
// Circle supports IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{
...
  public void Draw3D()
  { Console.WriteLine("Drawing Circle in 3D!"); }
}

// Hexagon supports IPointy and IDraw3D.
class Hexagon : Shape, IPointy, IDraw3D
{
...
  public void Draw3D()
  { Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

```
// Будет рисовать любую фигуру, поддерживающую IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
  Console.WriteLine("-> Drawing IDraw3D compatible type");
  itf3d.Draw3D();
}
```

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle(), new Circle("JoJo") } ;
  for(int i = 0; i < myShapes.Length; i++)
  {
    ...
    // Можно ли нарисовать эту фигуру в трехмерном виде?
    if(myShapes[i] is IDraw3D)
      DrawIn3D((IDraw3D)myShapes[i]);
  }
}
```

- Якщо тепер визначити, що приймає інтерфейс IDraw3D у якості параметра, то йому можна буде передавати, по суті, будь-який об’єкт, що реалізує IDraw3D.
  - При спробі передати тип, який не підтримує необхідний інтерфейс, компілятор повідомить про помилку.
  - У тривимірному вигляді відображається тільки об’єкт Hexagon, оскільки решта членів масиву Shape не реалізують інтерфейс IDraw3D.

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

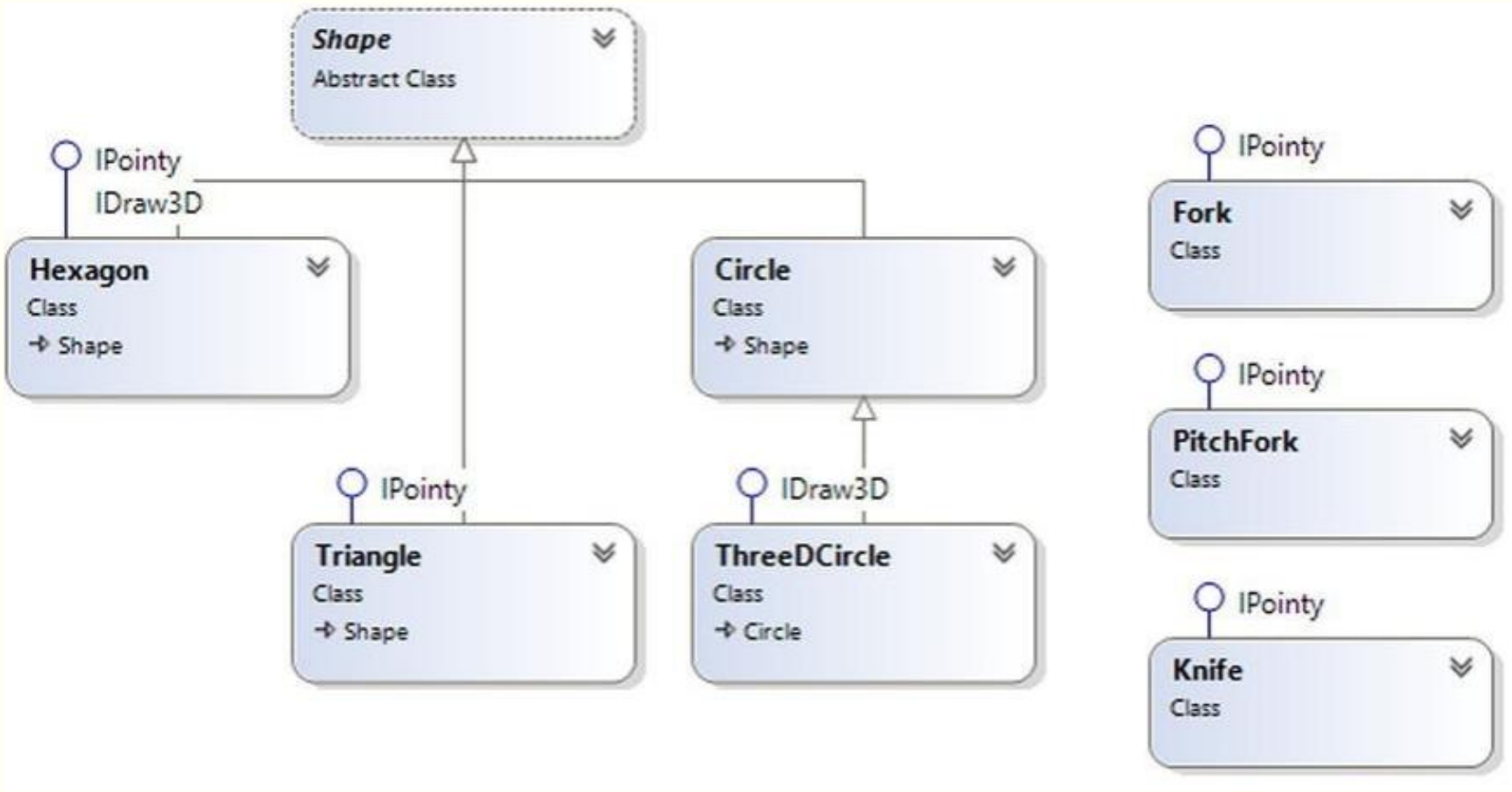
Застосування інтерфейсів у якості вихідних типів

```
// Этот метод возвращает из массива первый объект,
// который реализует интерфейс IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
  foreach (Shape s in shapes)
  {
    if (s is IPointy)
      return s as IPointy;
  }
  return null;
}
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  // Создать массив элементов Shape.
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")};
  // Получить первый элемент, имеющий вершины.
  // Для безопасности не помешает проверить firstPointyItem на предмет null.
  IPointy firstPointyItem = FindFirstPointyShape(myShapes);
  Console.WriteLine("The item has {0} points", firstPointyItem.Points);
  ...
}
```

- Для прикладу напишемо метод, який отримує масив об’єктів Shape і повертає посилання на перший елемент, який підтримує IPointy.

Масиви інтерфейсних типів

- Нехай у поточному проекті створені 3 нових класи, два з яких (Knife (ніж) і Fork (виделка)) моделюють кухонні прибори, а третій (PitchFork (вила)) — садовий інструмент

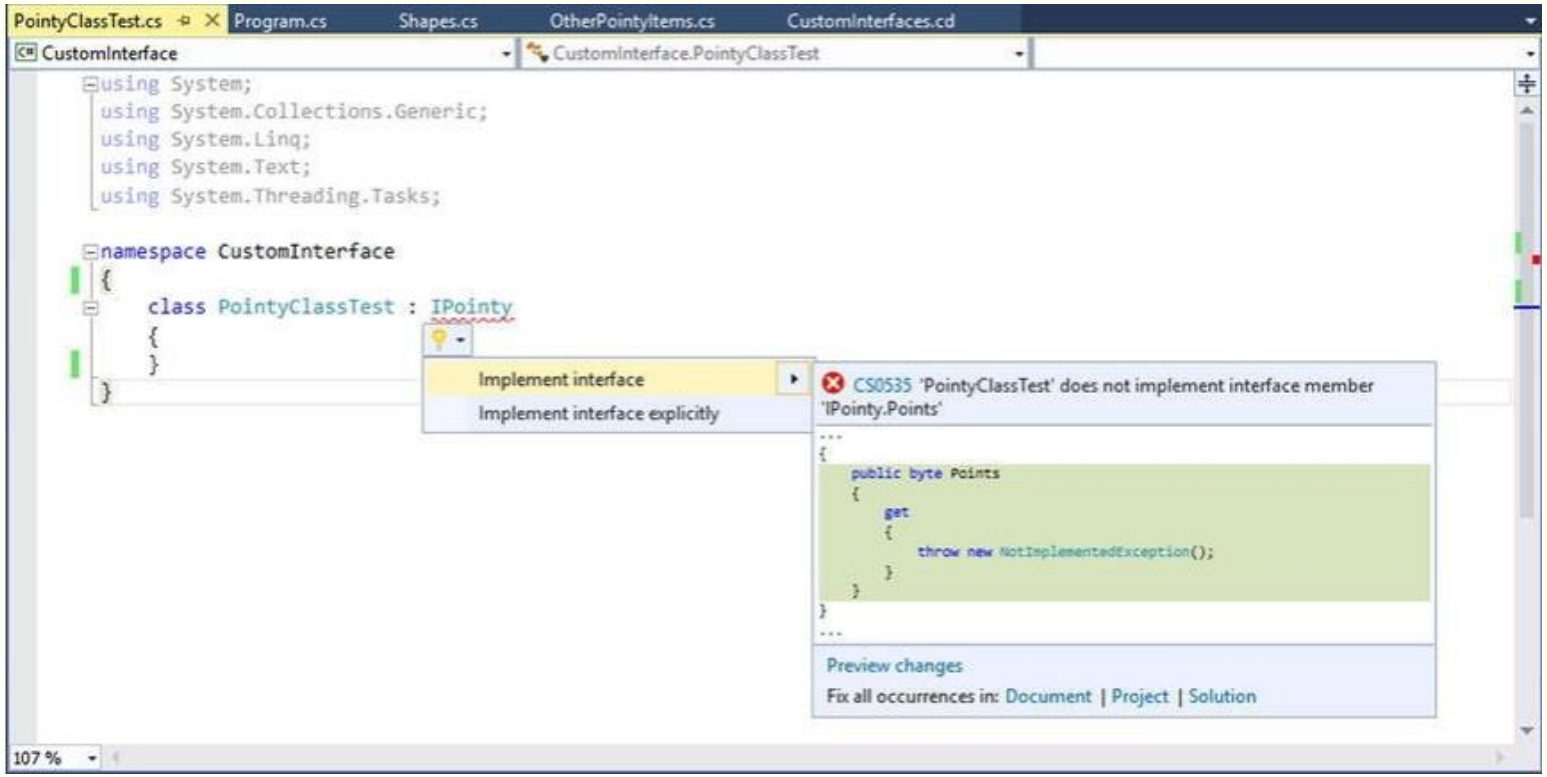


```
static void Main(string[] args)
{
  ...
  // Этот массив может содержать только типы,
  // которые реализуют интерфейс IPointy.
  IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
    new Triangle(), new Fork(), new PitchFork()};
  foreach(IPointy i in myPointyObjects)
    Console.WriteLine("Object has {0} points.", i.Points);
  Console.ReadLine();
}
```

- Маючи визначення типів PitchFork, Fork і Knife, можна визначити масив об’єктів, сумісних з IPointy.
  - Оскільки всі ці члени підтримують один і той же інтерфейс, можна виконувати прохід по масиву і інтерпретувати кожен його елемент як сумісний з IPointy об'єкт, незважаючи на різницю між ієрархіями класів.
  - масив заданого інтерфейсу може містити будь-який клас або структуру, яка реалізує цей інтерфейс.

Реалізація інтерфейсів за допомогою Visual Studio

- Для кожного методу інтерфейсу в кожному типі, що підтримує цю поведінку, потрібно вводити означення та реалізацію.
  - Для підтримки інтерфейсу, який визначає 5 методів і 3 властивості, необхідно приділяти увагу всім вісьмом членам, інакше виникатимуть помилки компіляції.
  - Відповідно до формальної термінології, назва забезпечена міткою — ***смарт-тегом***.



Поки оберемо перший варіант

- Visual Studio згенерирує код заглушки, призначений для подльшого оновлення.
  - Зверніть увагу, що стандартна реалізація генерує виняток System.NotImplementedException, що очевидно буде видалятись.

```
namespace CustomInterface
{
  class PointyTestClass : IPointy
  {
    public byte Points
    {
      get { throw new NotImplementedException(); }
    }
  }
}
```

Явна реалізація інтерфейсів

```
// Вывести изображение на форме.
public interface IDrawToForm
{
  void Draw();
}
// Вывести изображение в буфер памяти.
public interface IDrawToMemory
{
  void Draw();
}
// Вывести изображение на принтер.
public interface IDrawToPrinter
{
  void Draw();
}
```

- Завжди існує можливість реалізації інтерфейсів з членами, що мають ідентичні назви, тому виникає необхідність у вирішенні конфліктів імен.
  - Якщо тепер потрібно підтримувати всі ці інтерфейси в одному класі Octagon, компілятор дозволить використовувати визначення.
  - Хоч компіляція пройде гладко, присутня можлива проблема: надання єдиної реалізації методу Draw() не дозволяє робити унікальні дії на основі того, який інтерфейс отриманий від об'єкта Octagon.

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  public void Draw()
  {
    // Разделяемая логика вывода.
    Console.WriteLine("Drawing the Octagon...");
  }
}
```

```
static void Main(string[] args)
{
   Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
   // All of these invocations call the
   // same Draw() method!
   Octagon oct = new Octagon();

   IDrawToForm itfForm = (IDrawToForm)oct;
   itfForm.Draw();

   IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
   itfPriner.Draw();

   IDrawToMemory itfMemory = (IDrawToMemory)oct;
   itfMemory.Draw();

   Console.ReadLine();
}
```

Явна реалізація інтерфейсів

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  // Явно привязать реализации Draw()
  // к конкретным интерфейсам.
  void IDrawToForm.Draw()
  {
    Console.WriteLine("Drawing to form...");
  }
  void IDrawToMemory.Draw()
  {
    Console.WriteLine("Drawing to memory...");
  }
  void IDrawToPrinter.Draw()
  {
    Console.WriteLine("Drawing to a printer...");
  }
}
```

При реалізації декількох інтерфейсів, які мають ідентичні члени, вирішити конфлікт імен можна з використанням синтаксису явної реалізації інтерфейсів.

- при використанні цього синтаксису не вказується модифікатор доступу;
- явно реалізовані члени автоматично є ***закритими***.

```
// Ошибка! Модификатор доступа не может быть указан!
public void IDrawToForm.Draw()
{
  Console.WriteLine("Drawing to form...");
}
```

Явна реалізація інтерфейсів

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
  Octagon oct = new Octagon();

  // We now must use casting to access the Draw()
  // members.
  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();

  // Shorthand notation if you don't need
  // the interface variable for later use.
  ((IDrawToPrinter)oct).Draw();

  // Could also use the "is" keyword.
  if(oct is IDrawToMemory)
    ((IDrawToMemory)oct).Draw();

  Console.ReadLine();
}
```

- Оскільки явно реалізовані члени завжди неявно закриті, вони перестають бути доступними на рівні об'єктів.
  - якщо ви застосуєте до типу Octagon операцію доступу «.», то побачите, що IntelliSense не відображає ніяких членів Draw().
  - явну реалізацію інтерфейсів можна застосовувати і просто для приховування більш "складних" членів на рівні об'єктів.
  - У такому випадку при використанні операції «.» користувач об'єкта буде бачити тільки підмножину всієї функціональності типу.

Проєктування ієрархій інтерфейсів

- Як і в ієрархії класів, коли інтерфейс розширює існуючий інтерфейс, він успадковує всі абстрактні члени свого батька (або батьків).
  - похідний інтерфейс просто розширює власне визначення додатковими абстрактними членами.
  - Ієрархія інтерфейсів може бути зручна, коли потрібно розширити функціональність певного інтерфейсу без порушення роботи існуючих кодових баз.
- Спроектуємо новий набір інтерфейсів, пов'язаних з візуалізацією, так, щоб IDrawable був кореневим інтерфейсом в дереві цього сімейства:

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

  - Враховуючи, що IDrawable визначає базову поведінку малювання, можна створити похідний інтерфейс, який розширює IDrawable можливістю візуалізації в інших форматах, наприклад:

```
public interface IAdvancedDraw : IDrawable
{
  void DrawInBoundingBox(int top, int left, int bottom, int right);
  void DrawUpsideDown();
}
```

Проєктування ієрархій інтерфейсів

```
public class BitmapImage : IAdvancedDraw
{
  public void Draw()
  {
    Console.WriteLine("Drawing...");
  }
  public void DrawInBoundingBox(int top, int left, int bottom, int right)
  {
    Console.WriteLine("Drawing in a box...");
  }
  public void DrawUpsideDown()
  {
    Console.WriteLine("Drawing upside down!");
  }
}
```

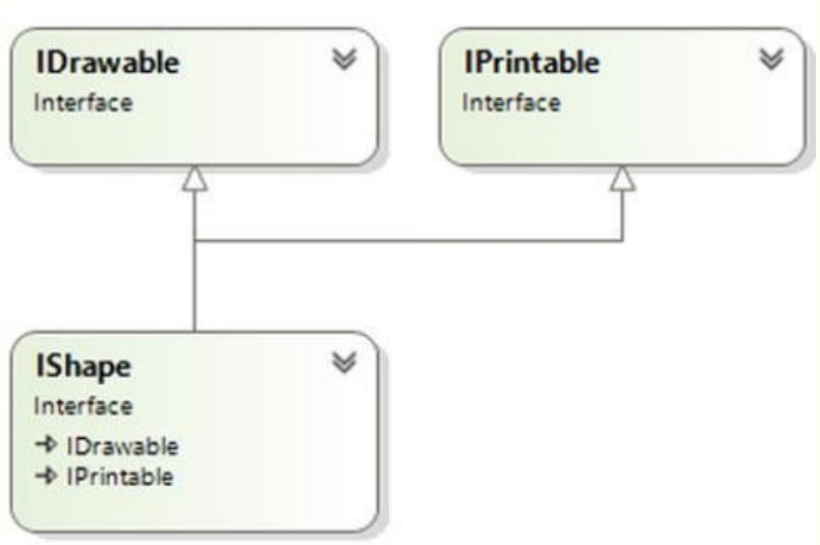
```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Interface Hierarchy *****");
  // Вызвать на уровне объекта.
  BitmapImage myBitmap = new BitmapImage();
  myBitmap.Draw();
  myBitmap.DrawInBoundingBox(10, 10, 100, 150);
  myBitmap.DrawUpsideDown();
  // Получить IAdvancedDraw явным образом.
  IAdvancedDraw iAdvDraw = myBitmap as IAdvancedDraw;
  if(iAdvDraw != null)
    iAdvDraw.DrawUpsideDown();

  Console.ReadLine();
}
```

- При такому проектному рішенні для реалізації інтерфейсу IAdvancedDraw в класі потрібно реалізувати всі члени, визначені в ланцюжку наслідування.
  - методи Draw(), DrawInBoundingBox() і DrawUpsideDown().
- Тепер у разі використання BitmapImage можна викликати кожен метод на рівні об'єкта (тому що всі вони є відкритими), а також отримувати посилання на кожен підтримуваний інтерфейс явно за допомогою зведення.

Множинне наслідування за допомогою інтерфейсних типів

```
// Множественное наследование для интерфейсных типов разрешено
interface IDrawable
{
   void Draw();
}
interface IPrintable
{
   void Print();
   void Draw();  // <-- Здесь возможен конфликт имен!
}
// Множественное наследование интерфейсов. Нормально!
interface IShape : IDrawable, IPrintable
{
   int GetNumberOfSides();
}
```

- На відміну від класів, один інтерфейс може розширювати відразу кілька базових інтерфейсів, що дозволяє проектувати дуже потужні й гнучкі абстракції.



```
class Rectangle : IShape
{
  public int GetNumberOfSides()
  { return 4; }

  public void Draw()
  { Console.WriteLine("Drawing..."); }

  public void Print()
  { Console.WriteLine("Prining..."); }
}
```

```
class Square : IShape
{
  // Использование явной реализации
  void IPrintable.Draw()
  {
    // Вывести на принтер...
  }
  void IDrawable.Draw()
  {
    // Вывести на экран...
  }
  public void Print()
  {
    // Печатать...
  }
  public int GetNumberOfSides()
  { return 4; }
}
```

Головне питання: скільки методів повинен реалізовувати клас, що підтримує Ishape?

- Відповідь: залежно від обставин.
  - Якщо потрібно надати просту реалізацію методу Draw(), потрібно тільки реалізувати три його члена, як у класі Rectangle.
  - Якщо краще мати специфічні реалізації для кожного методу Draw() (тут має сенс), конфлікт імен можна вирішити із застосуванням явної реалізації інтерфейсів, як у класі Square.
- вже зараз важливо усвідомити, що інтерфейси є фундаментальним аспектом .NET Framework.
  - Незалежно від типу, який розробляється (веб-додаток, настільний додаток з графічним інтерфейсом користувача, бібліотека доступу до даних і т.п.), робота з інтерфейсами буде складовою частиною цього процесу.

Методи за умовчанням в інтерфейсах (починаючи з .NET Core 3.0)

- До C# 8.0 інтерфейси могли вміщати лише оголошення методів (без реалізації), а члени інтерфейсу були публічними та абстрактними за умовчанням.
  - Також інтерфейс не міг містити полів чи закритих/захищених/внутрішніх членів.
  - При представленні нового члена інтерфейсу всі класи, які реалізують цей інтерфейс потрібно було оновлювати.
- У C# 8.0 вводяться інтерфейсні методи за умовчанням (default interface methods).
  - Також члени інтерфейсів тепер можуть бути private, protected та static.
  - До захищених членів інтерфейсу не можна отримати доступ з класу, який реалізує інтерфейс – доступ є у породжених інтерфейсів.
  - Члени інтерфейсів можуть бути також virtual і abstract.
  - Проте віртуальні члени інтерфейсу теж можуть заміщатись у породженому інтерфейсі, а не в реалізуючому класі.
- instance member все ще не можна додавати в інтерфейси.

Навіщо методи за умовчанням в інтерфейсах?

```
public interface ILogger
{
    public void Log(string message);
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // деякий код
    }
}
public class DbLogger : ILogger
{
    public void Log(string message)
    {
        // деякий код
    }
}
```

- Інтерфейсні методи за умовчанням містять конкретні реалізації.
  - Якщо клас, який реалізує інтерфейс, не містить власної реалізації методу, буде застосована реалізація за умовчанням з інтерфейсу.
  - Це допомагає безпечно додавати методи в інтерфейс, не порушуючи роботу існуючої функціональності.
- Нехай потрібно представити новий метод в інтерфейс ILogger, який прийматиме 2 параметри (текстове повідомлення та рівень логу).

Навіщо методи за умовчанням в інтерфейсах?

```
public enum LogLevel
{
    Info, Debug, Warning, Error
}
public interface ILogger
{
    public void Log(string message);
    public void Log(string message, LogLevel logLevel)
    {
        Console.WriteLine("Log method of ILogger called.");
        Console.WriteLine("Log Level: "+ logLevel.ToString());
        Console.WriteLine(message);
    }

}
```

- Проблема: ви змушені реалізовувати новий метод у всіх класах, що реалізують інтерфейс ILogger.
  - Інакше компілятор сигналізує про помилку.
  - Інтерфейс може використовуватись у кількох інших бібліотеках та навіть між командами, тому така зміна буде болісною.
- Класи, які реалізують інтерфейс ILogger, тепер не вимагають впровадження нового метода Log().
  - Попередній код нормально компілюється.

Класи, які реалізують інтерфейс, не знають про інтерфейсні методи за умовчанням

- Створимо екземпляр класу FileLogger:

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        ILogger logger = new FileLogger();
        logger.Log("This is a test message.", LogLevel.Debug);

        FileLogger fileLogger = new FileLogger();
        fileLogger.Log("This is a test message.", LogLevel.Debug);

        Console.Read();
    }
}
```

void FileLogger.Log(string message)

No overload for method 'Log' takes 2 arguments

Show potential fixes (Alt+Enter or Ctrl+.)

Множинне наслідування інтерфейсів та інтерфейсні методи за умовчанням

```
public interface A
{
    public void Display();
}
public interface B : A
{
    public void Display()
    {
        Console.WriteLine("Interface B.");
    }
}
public interface C : A
{
    public void Display()
    {
        Console.WriteLine("Interface C.");
    }
}
public class MyClass : B, C
{
}
```

- При компіляції коду зліва отримаємо помилку: клас MyClass не реалізує член інтерфейсу A.Display().
  - Необхідно забезпечити реалізацію, щоб задовольнити компілятор:

```
public class MyClass : B, C
{
    public void Display()
    {
        Console.WriteLine("MyClass.");
    }
}
static void Main(string[] args)
{
    A obj = new MyClass();
    obj.Display();
    Console.Read();
}
```

  - Для усунення неоднозначності застосовується найбільш конкретна реалізація – з класу MyClass.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Вбудовані інтерфейси .NET