Язык C#

Интерфейсы

Лекция #3

Определение интерфейса

- Интерфейс – это набор семантически связанных абстрактных методов, свойств и событий.
- Класс, реализующий интерфейс, должен самостоятельно полностью переопределить все члены данного интерфейса.

Пример интерфейса

```
// Этот интерфейс определяет возможности
// работы с углами геометрической фигуры
public interface IPointy
{
 // Автоматически этот член интерфейса становится абстрактным
  byte GetNumberOfPoints();

  // Чтобы сделать это свойство «только для чтения»
  // достаточно просто удалить блок set
  // byte Points(get; set;)

}
```

Пример интерфейса

```
// Любой класс может реализовывать любое количество интерфейсов, но он должен
// производиться только от одного базового класса:
public class Hexagon : Shape, IPointy
{
    public Hexagon(){}
    public Hexagon(string name): base(name){}

    public override void Draw()
    {
        // Вспомним, что в классе Shape определено свойство PetName
        Console.WriteLine("Drawing {0} the Hexagon", PetName);
    }

    // Реализация IPointy
    public byte GetNumberOfPoints()
    {
        return 6;
    }
}
```

Пример интерфейса

```
public 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 GetNumberOfPoints()
    {
        return 3;
    }
}
```

В классе должны быть реализованы все методы интерфейса, половинчатого решения быть не может

Получение ссылки на интерфейс

```
// Получаем ссылку на интерфейс IPointy,
// используя явное приведение типов

Hexagon hex = new Hexgon("Bill");
IPointy itfPt = (IPointy) hex;
Console.WriteLine (itfPt.GetNumberOfPoints());


// Перехват исключения
Circle c = new Circle("Lisa");
IPointy itfPt;
try {  itfPt = (IPointy) c;
    Console.WriteLine (itfPt.GetNumberOfPoints());
     }
catch (IvalidCastException e)
    {  Console.WriteLine("Oops! Not pointy…");
    }
```

Получение ссылки на интерфейс с помощью оператора **as**

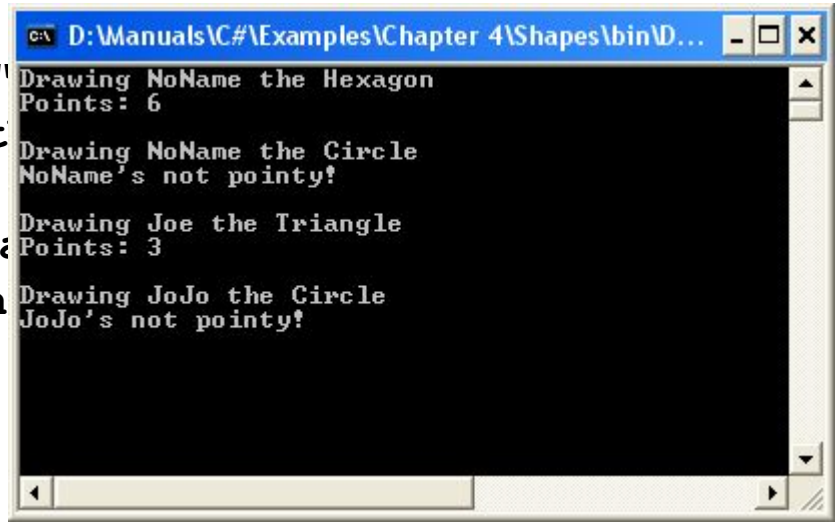
```
// Еще один способ получить ссылку на интерфейс

Hexagon hex = new Hexgon("Bill");
IPointy itfPt;
itfPt = hex as IPointy;
Console.WriteLine (itfPt.GetNumberOfPoints());

if (itfPt != null)
  Console.WriteLine(itfPt.GetNumberOfPoints());
else
  Console.WriteLine("Oops! Not pointy…");
```

Проверка существования интерфейса с помощью оператора **is**

```
// Есть ли у этой фигуры углы?

Triangle t = new Triangle();

if (t is IPointy)
  Console.WriteLine(t.GetNumberOfPoints());
else
  Console.WriteLine("Oops! Not pointy…");
```

Пример использования **is**

```
// Давайте выясним (во время выполнения), у каких
// геометрических фигур есть углы
Shape[] s = {new Hexagon(), new Circle(),
                new Triangle("Joe"), new Circle("JoJO")}

for(int i = 0; i < s.length; i++)
{
   // Вспомним, что базовый класс Shape() определяет
   // абстрактный метод Draw()
    s[i].Draw();

   // У каких геометрических фигур в массиве есть углы?
   if (s[i] is IPointy)
    Console.WriteLine("Points: {0}"
                ((IPointy)s[i]).Get
    else
        Console.WriteLine(s[i].PetNa
                        "\'s not poin
}
```



Интерфейсы как параметры

```
// Интерфейс для отображения фигур в трех измерениях
public interface IDraw3D  {  void Draw3D(); }

// Circle поддерживает интерфейс IDraw3D
public class Circle : Shape, IDraw3D
{ ...
    public void Draw3D()
    {
      Console.WriteLine("Drawing Circle in 3D!");
    }
}

// Если наши типы поддерживают несколько интерфейсов, нужно
// просто перечислить эти интерфейсы через запятую, как обычно:
public class Hexagon : Shape, IPointy, IDraw3D
{ ...
    public void Draw3D()
    {
      Console.WriteLine("Drawing Hexagon in 3D!");
    }
}
```

Интерфейсы как параметры

```
// Создаем несколько геометрических фигур. Если они поддерживают
// отображение  в трех измерениях, делаем это!
public class ShapesApp
{
// Будут нарисованы все объекты, поддерживающие интерфейс IDraw3D
  public static void DrawThisShapeIn3D(IDraw3D itf3d)
  {
    itf3d.Draw3D();
  }

  public static int Main(string[] args)
  {
    Shape[] s = {new Hexagon(), new Circle(),                    new
Triangle(), new Circle("JoJo")};

    for(int i=0; i<s.Length; i++)
    {
    // Могу ли я нарисовать этот объект в трех измерениях?
        if(s[i] is IDraw3D)
            DrawThisShapeIn3D((IDraw3D)s[i]);
    }
    return 0;
  }
}
```

Разрешение конфликтов имен

```
public interface IDraw3D
{  void Draw3D();
}

// А что будет, если  мы сделаем
public interface IDraw3D
{
  void Draw();
}

public class Line : Shape, IDraw3D
// И базовый класс и интерфейс определяют метод Draw
{
  public override void Draw()
  {
    Console.WriteLine("Drawing a line…");
  }
}
```

Разрешение конфликтов имен

```
// Вызываем Line.Draw()

Line myLine = new Line();
myLine.Draw()


// Вызываем Line.Draw() еще раз, но уже по-другому

IDraw3D itfDraw3D = (IDraw3D) myLine;
itfDraw3D.Draw();

// В обоих случаях будет вызван один и тот же метод!
```

Явная реализация интерфейса

```
// При помощи явной реализации методов интерфейса мы можем
// определить  разные варианты метода Draw()
public class Line : Shape, IDraw3D
{
    // Этот метод можно будет вызвать только через
    // ссылку на интерфейс IDraw3D
    void IDraw3D.Draw()
    {
        Console.WriteLine("Drawing a 3D line...");
    }

    // Этот метод можно будет вызвать только через
    // ссылку на объект класса Line
    public override void Draw()
    {
        Console.WriteLine("Drawing a line...");
    }
}
```

Нельзя использовать модификаторы области видимости для методов интерфейса

Явная реализация интерфейса

```
// Конфликтов имен не будет!
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
    void IDraw.Draw()
    {
        // Вывод обычного плоского изображения
    }

    void IDrawToPrinter.Draw()
    {
        // Вывод на принтер
    }

    void IDraw3D.Draw()
    {
        // Поддержка объемного изображения
    }
}
```

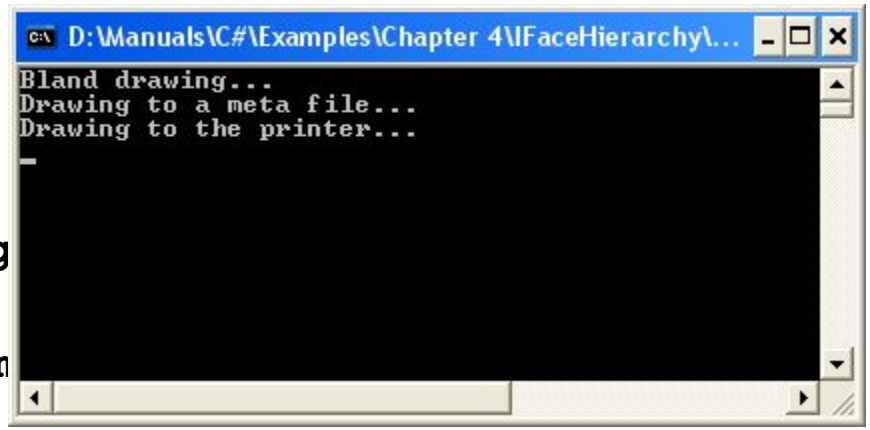
Создание иерархий интерфейсов

```
// Базовый интерфейс
interface IDraw
{
   void Draw();
}

interface IDraw2 : IDraw
{
  void  IDrawToPrinter();
}

interface IDraw3 : IDraw2
{
  void  IDrawToMetafile();
}
```

Создание иерархий интерфейсов

```
// Этот класс будет поддерживать IDraw, IDraw2 и IDraw3
public class SuperImage : IDraw3
{
// Используем явную реализацию интерфейсов, чтобы привязать
// методы  к конкретным интерфейсам
    void IDraw.Draw()
    {
        // Обычный вывод на экран
    }

    void IDraw2.DrawToPrinter()
    {
        // Вывод на принтер
    }

    void IDraw3.DrawToMetafile()
    {
        // Вывод в метафайл
    }
}
```

Создание иерархий интерфейсов




```
// Проверяем наши интерфейсы
public class TheApp
{
 public static int Main(string
 {
    SuperImage si = new SuperIm

    // Получаем ссылку на интерфейс IDraw
    IDraw itfDraw = (IDraw)si;
    itfDraw.Draw();

    // А теперь получаем ссылку на интерфейс IDraw3
    if(itfDraw is IDraw3)
    {
        IDraw3 itfDraw3 = (IDraw3)itfDraw;
        itfDraw3.DrawToMetaFile();
        itfDraw3.DrawToPrinter();
    }
    return 0;
  }
}
```

Наследование от нескольких базовых интерфейсов

```
    interface IBasicCar
    {   void Drive();
    }

    interface IUnderwaterCar
    {   void Dive();
    }

    interface IJamesBondCar : IBasicCar, IUnderwaterCar
    {   void TurboBoost();
    }

public class JBCar : IJamesBondCar
{   public JBCar(){}

    // Унаследованные члены
    void IBasicCar.Drive()
        {Console.WriteLine("Speeding up...");}
    void IUnderwaterCar.Dive()
        {Console.WriteLine("Submerging...");}
    void IJamesBondCar.TurboBoost()
        {Console.WriteLine("Blast off!");}
}
```

Наследование от нескольких базовых интерфейсов

```
JBCar j = new JBCar();
if(j is IJamesBondCar)
{
    ((IJamesBondCar)j).Drive();
    ((IJamesBondCar)j).TurboBoost();
    ((IJamesBondCar)j).Dive();
}
```

Использование встроенных интерфейсов

Интрефесы IEnumerate, IEnumerate

Описание класса Car

```
// Cars – набор объектов класса Car
public class Cars
{
   private Car[] carArray;

// При создании объекта класса Cars заполняем его несколькими
// объектами Car
   public Cars()
   {
    carArray = new Car[4];
    carArray[0] = new Car("FeeFee", 200, 0);
    carArray[1] = new Car("Clunker", 90, 0);
    carArray[2] = new Car("Zippy", 30, 0);
    carArray[3] = new Car("Fred", 30, 0);
   }
}
```

Применить foreach?

```
// Кажется очень заманчивым
public class CarDriver
{
  public static void Main()
  {
     Cars carLot = new Cars();
     // Пробуем использовать foreach для обращения
     // к каждому объекту Car внутри  набора,
     //  представленного carLot
     foreach (Car c in carLot)
   {
     Console.WriteLine("Name: {0}", c.PetName);
     Console.WriteLine("Max speed: {0}", c.MaxSpeed);
   }
  }
}
```

Класс Cars не реализует метод GetEnumerator();

Определяется в IEnumerable в System.Collection

Interface IEnumerable

```
// Необходимо, чтобы класс реализовывал
// интерфейс IEnumerable
public class Cars : IEnumerable
{
  …
  // Интерфейс IEnumerable определяет этот метод
  // и ничего больше!
  public IEnumerator GetEnumerator()
  {
    // А дальше-то что?
  }
  …
}
```

Интерфейс IEnumerator

```
public interface IEnumerator
{
  // Передвинуть внутренний указатель на одну позицию
  bool MoveNext();

 // Получить текущий элемент набора
  object Current (get;)

 // Установить внутренний указатель на начало набора
  void Reset();
}
```

Модификация класса Cars

```
// Набор объектов Car с реализованным нумератором!
public class Cars: IEnumerator, IEnumerable
{
   private car[ ] carArray;
   int pos = -1;   // Переменная для текущей позиции элемента в массиве

   public Cars()
   { // Здесь мы создаем несколько объектов класса Car и добавляем их в массив
   }

   public bool MoveNext() // Реализация методов интерфейса IEnumerator
   {
   if (pos < carArray.Length) { pos++; return true; }
   else return false;
   }

   public void Reset() { pos = 0; }

   public object Current
   {  get { return carArray[pos]; }
   }
    // Реализация метода интерфейса IEnumerable
   public IEnumerator GetEnumerator()
   {     return (IEnumerator) this;
   }
}
```

Применить foreach!

```
public class CarDriver
{
  public static void Main()
  {
     Cars carLot = new Cars();
     // Теперь можно использовать foreach для обращения
     // к каждому объекту Car внутри  набора
     foreach (Car c in carLot)
   {
     Console.WriteLine("Name: {0}", c.PetName);
     Console.WriteLine("Max speed: {0}", c.MaxSpeed);
   }
  }
}
```

Дополнительные способы обращения к объектам Car

```
// Обращаемся к объектам Car через IEnumerator
IEnumerator itfEnum;
itfEnum = (IEnumerator)carLot;

// Устанавливаем курсор на начало
itfEnum.Reset();

// Перемещаем курсор вперед на один шаг
itfEnum.MoveNext();

// Выбираем одну машину и включаем в ней радио
object curCar = itfEnum.Current;
((Car)curCar).CrankTunes(true);
```

Создание клонируемых объектов

Интерфейс ICloneable

Поверхностное копирование (shallow copy)

```
// Наш класс — это просто точка с координатами на плоскости
public class Point
{
   // Поля (открытые переменные)
   public int x, y;

   // Конструкторы
   public Point() {}
   public Point(int x, int y){this.x = x; this.y = y;}

   // Замещаем Object.ToString()
   public override string ToString()
   {return "X: " + x + " Y: " + y; }
}
```

Оператор = вызывает метод MemberwiseClone();

Глубокое копирование (deep copy)

```
// Реализуем в классе Point поддержку глубокого копирования
// через интерфейс ICloneable
public class Point : ICloneable
{
   // Данные о состоянии объекта
   public int x, y;

   // Конструкторы
   public Point() {}
   public Point(int x, int y){this.x = x; this.y = y;}

   // Реализуем единственный метод ICloneable
   public object Clone()
   {
    return new Point(this.x, this.y)
   }

   public override string ToString()
   {return "X: " + x + " Y: " + y; }
}
```

Глубокое копирование (deep copy)

```
// Обратите внимание, что Clone() возвращает
// "объект вообще". Чтобы получить из него
// нужный нам производный тип, придется провести
// явное преобразование типов

Point p3 = new Point(100, 100);
Point p4 = (Point) p3.Clone();

// Меняем p4.x (при этом p3.x не изменится)
p4.x = 0;

// Проверяем, так ли это:
Console.WriteLine("Deep copying using Clone()");
Console.WriteLine(p3);
Console.WriteLine(p4)
```

Сравнивание объектов

Интерфейс IComparable

Интерфейс IComparable

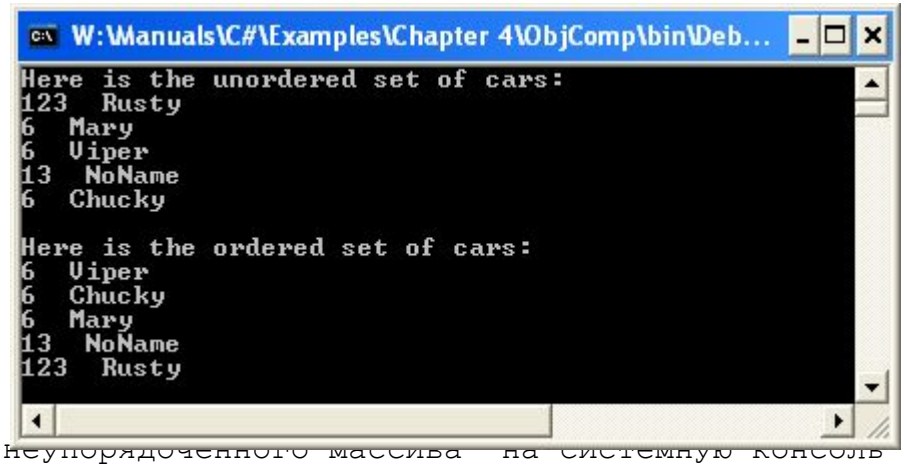
```
Car[] myAutos = new Car[5];
myAutos[0] = new Car(123, "Rusty");
myAutos[1] = new Car(6, "Mary");
myAutos[2] = new Car(83, "Viper");
myAutos[3] = new Car(13, "NoName");
myAutos[4] = new Car(9873, "Chucky");

Array.Sort(myAutos); // Что-то не выходит

// Этот интерфейс позволяет определить место объекта
// среди других аналогичных объектов
interface IComparable
{
  int CompareTo(object o)
}
```

Реализация интерфейса IComparable

```
// Такая реализация метода CompareTo() позволит сортировать
// объекты автомобилей по значению идентификатора – CarID
public class Car : IComparable
{
...
   // Реализация IComparable
   int IComparable.CompareTo(object o)
   {
    Car temp = Car(o);
    if(this.CarID > temp.CarID)
        return 1;
    if(this.CarID < temp.CarID)
        return -1;
    else
        return 0;
   }
}
```

Применяем интерфейс IComparable

```
// Применяем реализованный нами интерфейс IComparable на практике
public class CarApp
{
  public static int Main(string[] args)
  {
  // Создаем массив объектов Car
  Car[] myAutos = new Car[5];
  myAutos[0] = new Car(123, "Rusty");
  myAutos[1] = new Car(6, "Mary");
  myAutos[2] = new Car(83, "Viper");
  myAutos[3] = new Car(13, "NoName");
  myAutos[4] = new Car(9873, "Chucky");

  // Выводим информацию об автомобилях из неупорядоченного массива на системную консоль
  Console.WriteLine("Here is the unordered set of cars:");
  foreach(Car c in MyAutos)
     Console.WriteLine(c.ID + " " + c.PetName);

  // А теперь используем возможности только что реализованного IComparable
  Array.Sort(myAutos);

  // Выводим информацию уже из упорядоченного массива
  Console.WriteLine("Here is the ordered set of cars:");
  foreach(Car c in myAutos)
     Console.WriteLine(c.ID + " " c.PetName);
    return 0;
 }
}
```



Сортировка по нескольким идентификаторам Интерфейс IComparer в System.Collection

```
// Стандартный способ сравнения двух объектов
interface IComparer
{
  int Compare(object o1, object o2)
}
```

Создание вспомогательного класса

```
// Этот вспомогательный класс нужен для
// сортировки объектов Car по PetName
using System.Collections;
public class SortByPetName : IComparer
{
  public SortByPetName(){}

  // Сравниваем прозвища (PetName) объектов
  int IComparer.Compare(object o1, object o2)
  {
   Car t1 = (Car)o1;
   Car t2 = (Car)o2;
   return String.Compare(t1.PetName, t2.PetName);
  }
}
```

Сортировка по указанному полю

```
// Now sort by pet name.
Array.Sort(myAutos, new SortByPetName());

Console.WriteLine("\nOrdering by pet name:");
foreach(Car c in myAutos)
  Console.WriteLine(c.ID + "  " + c.PetName);
```

Можно включить статическое свойство

```
public class Car : IComparable
{
// As a nested class!
private class SortByPetNameHelper : IComparer
{  public SortByPetNameHelper(){}

  // IComparer impl.
  int IComparer.Compare(object o1, object o2)
  { Car t1 = (Car)o1;
   Car t2 = (Car)o2;
   return String.Compare(t1.PetName, t2.PetName);
    }
 }
  // Property to return the SortByPetName comparer.
  public static IComparer SortByPetName
   { get { return (IComparer)new SortByPetNameHelper(); } }

// Теперь можно так
Array.Sort(myAutos, Car.SortByPetName);
```

Пространство имен System.Collections

Интерфейсы и классы

Интерфейсы

ICollections	Общие характеристики
IComparer	Сравнение двух объектов
IDictionary	Представление объекта в виде пары значений
IDictionaryEnumerator	Нумерация содержимого объекта, поддерживающего IDictionary
IEnumerable	Возвращает IEnumerator для указанного объекта
IEnumerator	Используется для поддержки foreach
IHashCodeProvider	Возвращает хэш-код для реализации типа
IList	Обеспечивает методы для добавления, удаления и индексирования в списке объектов

Классы

ArrayList	Динамически изменяющий свой размер массив объектов
HashTable	Представляет набор взаимосвязанных ключей и значений
Queue	Стандартная очередь по принципу FIFO
SortedList	Аналогично словарю, но к элементам можно обращаться по их порядковому номеру
Stack	Очередь, организованная по принципу LIFO

Применение ArrayList

```
// Нам больше не нужно реализовывать IEnumerator — все уже сделано за нас в ArrayList
public class Cars : IEnumerable
{
 // Это — тот самый внутренний класс, который и будет делать всю работу
 private ArrayList carList;

 // Создаем объект класса carList при помощи конструктора Cars
 public Cars() {carList = new ArrayList();}

 // Реализуем нужные нам методы для приема вызовов извне и передачи их carList

 // Метод для вставки объекта Car
 public void AddCar(Car c) { carList.Add(c); }

 // Метод для удаления объекта Car
 public void RemoveCar(int carToRemove) { carList.RemoveAt(carToRemove); }

 // Свойство, возвращающее количество объектов Car
 public int CarCount { get { return carList.Count; } }

 // Метод для очистки объекта — удаления всех объектов Car
 public void ClearAllCars() { carList.Clear(); }

 // Метод, который отвечает на вопрос — есть ли уже в наборе такой объект Car
 public bool CarIsPresent(Car c) { return carList.Contains(c); }

 // А все, что связано с реализацией IEnumerator, мы просто перенаправляем в carList
 public IEnumerator GetEnumerator() { return carList.GetEnumerator(); }
}
```

Применение нового варианта Cars

```
public static void Main()
{  Cars carLot = new Cars();

  // Чтобы было с чем работать, добавляем несколько объектов Car
  carLot.AddCar( new Car("Jasper", 200, 80));
  carLot.AddCar( new Car("Mandy", 140, 80));
  carLot.AddCar( new Car("Porker", 90, 90));
  carLot.AddCar( new Car("Jimbo", 40, 4));

  // Выводим информацию о каждом классе при помощи конструкции foreach
  Console.WriteLine("You have {0} in the lot: \n", carLot.CarCount);
  foreach (Car c in carLot)
  {  Console.WriteLine("Name: {0}', c.PetName);
   Console.WriteLine("Max speed: {0}\n", c.MaxSpeed);
  }

  carLot.RemoveCar(3); // Удаляем одну из машин
  Console.WriteLine("You have {0} in the lot.\n", carLot.CarCount);

  // Добавляем еще одну машину и проверяем ее наличие в наборе
  Car temp = new Car("Zippy", 90, 90);
  carLot.AddCar(temp);

  if(carLot.CarIsPresent(temp))
   Console.WriteLine(temp.PetName + " is already in the lot.");

  carLot.ClearAllCars(); // Убить их всех!
  Console.WriteLine("You have {0} in the lot.\n", carLot.CarCount");
}
```