

# Курс «С#. Программирование на языке высокого уровня»

---

Павловская Т.А.

# Лекция 9. Интерфейсы. Контейнерные классы

---

**Описание и использование интерфейсов. Применение стандартных интерфейсов .NET для сравнения, перебора, сортировки и клонирования объектов. Понятие контейнера (коллекции). Использование стандартных коллекций .NET.**

---

# Интерфейсы

# Общие сведения об интерфейсе

- *Интерфейс* является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах.
- Интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами.
- Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.
- Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.
- Синтаксис интерфейса аналогичен синтаксису класса:  
[ атрибуты ] [ спецификаторы ] **interface** имя [ : предки ]  
тело\_интерфейса [ ; ]

- Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую.
- *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.
- Интерфейс не может содержать константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

# Область применения интерфейсов

- Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.
- То, что работает в пределах иерархии одинаково, предпочтительно полностью определить в базовом классе.
- Интерфейсы же чаще используются для задания общих свойств объектов различных иерархий.

# Отличия интерфейса от абстрактного класса

- элементы интерфейса по умолчанию имеют спецификатор доступа `public` и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;
- класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
- класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

# Реализация интерфейса

- В C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению.
- Сигнатуры методов в интерфейсе и реализации должны полностью совпадать.
- Для реализуемых элементов интерфейса в классе следует указывать спецификатор `public`.
- К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.

# Пример

## interface IAction

```
{ void Draw(); int Attack( int a ); void Die(); int Power { get; } }
```

## class Monster : IAction

```
{ public void Draw() { Console.WriteLine( "Здесь был " + name ); }  
  public int Attack( int ammo_ ) {  
    ammo -= ammo_;  
    if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" ); else ammo = 0;  
    return ammo;  
  }  
  public void Die()  
    { Console.WriteLine( "Monster " + name + " RIP" ); health = 0; }  
  public int Power { get { return ammo * health; } }  
}
```

```
Monster Vasia = new Monster( 50, 50, "Вася" ); // объект класса Monster  
  Vasia.Draw(); // результат: Здесь был Вася  
IAction Actor = new Monster( 10, 10, "Маша" ); // объект типа интерфейса  
  Actor.Draw(); // результат: Здесь был Маша
```

# Обращение к реализованному методу через объект типа интерфейса

- Удобство этого способа проявляется при присваивании объектам типа `IAction` ссылок на объекты различных классов, поддерживающих этот интерфейс.
- Например, есть метод с параметром типа интерфейса. На место этого параметра можно передавать любой объект, реализующий интерфейс:

```
static void Act( IAction A )
{
    A.Draw();
}
static void Main()
{
    Monster Vasia = new Monster( 50, 50, "Вася" );
    Act( Vasia );
    ...
}
```

# Второй способ реализации интерфейса

*Явное указание имени интерфейса* перед реализуемым элементом. Спецификаторы доступа не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*:

```
class Monster : IAction {  
    int IAction.Power { get{ return ammo * health;}}  
    void IAction.Draw() {  
        Console.WriteLine( "Здесь был " + name );    }  
}
```

...

```
IAction Actor = new Monster( 10, 10, "Маша" );  
Actor.Draw();           // обращение через объект типа интерфейса  
// Monster Vasia = new Monster( 50, 50, "Вася" );  
// Vasia.Draw();           ошибка!
```

При этом соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Кроме того, этот способ позволяет избежать конфликтов при множественном наследовании

# Пример

Пусть класс `Monster` поддерживает два интерфейса: один для управления объектами, а другой для тестирования:

```
interface Itest { void Draw(); }  
interface Iaction { void Draw(); int Attack( int a ); ... }  
class Monster : IAction, Itest {  
    void ITest.Draw() {  
        Console.WriteLine( "Testing " + name );    }  
    void IAction.Draw() {  
        Console.WriteLine( "Здесь был " + name );    }  
    ... }  
}
```

Оба интерфейса содержат метод `Draw` с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса.

Обращаются к этим методам, используя **операцию приведения типа**:

```
Monster Vasia = new Monster( 50, 50, "Вася" );  
((ITest)Vasia).Draw();           // результат: Здесь был Вася  
((IAction)Vasia).Draw();       // результат: Testing Вася
```

# Операция is

- При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс.
- Проверка выполняется с помощью бинарной операции is. Она определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа.
- Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте:

```
if ( объект is тип )
```

```
{  
    // выполнить преобразование "объекта" к "типу"  
    // выполнить действия с преобразованным объектом  
}
```

# Операция as

- Операция as выполняет преобразование к заданному типу, а если это невозможно, формирует результат null:

```
static void Act( object A )  
{  
    IAction Actor = A as IAction;  
    if ( Actor != null ) Actor.Draw();  
}
```

- Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

# Интерфейсы и наследование

- Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня.
- Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки.
- Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его.
- В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово `new`, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается.

# Пример

```
interface IBase { void F( int i ); }
interface Ileft : IBase {
    new void F( int i );          /* переопределение метода F */ }
interface Iright : IBase { void G(); }
interface Iderived : Ileft, Iright {}
class A {
    void Test( Iderived d ) {
        d.F( 1 );                // Вызывается Ileft.F
        ((IBase)d).F( 1 );       // Вызывается IBase.F
        ((Ileft)d).F( 1 );       // Вызывается Ileft.F
        ((Iright)d).F( 1 );      // Вызывается IBase.F
    }
}
```

Метод F из интерфейса IBase скрыт интерфейсом Ileft, несмотря на то, что в цепочке Iderived — Iright — IBase он не переопределялся.

# Особенности реализации интерфейсов

- Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент.
- Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса.
- Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса.

# Стандартные интерфейсы .NET

- В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс **IComparable** задает метод сравнения объектов на «больше-меньше», что позволяет выполнять их сортировку.
- Реализация интерфейсов **IEnumerable** и **IEnumerator** дает возможность просматривать содержимое объекта с помощью `foreach`, а реализация интерфейса **ICloneable** — клонировать объекты.
- Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью `foreach` возможна оттого что тип `Array` реализует интерфейсы `IEnumerable` и `IEnumerator`.
- Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

# Сравнение объектов

```
int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };  
Array.Sort(numbers);  
foreach (int n in numbers) Console.WriteLine(n);
```

- Интерфейс `IComparable` определен в пространстве имен `System`. Он содержит всего один метод `CompareTo`, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface IComparable
```

```
{
```

```
    int CompareTo( object obj )
```

```
}
```

- Метод должен возвращать:
  - 0, если текущий объект и параметр равны;
  - отрицательное число, если текущий объект меньше параметра;
  - положительное число, если текущий объект больше параметра.

# Интерфейс IComparable

```
class Person : IComparable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int CompareTo(object o)
    {
        Person p = o as Person;
        if (p != null)
            return this.Name.CompareTo(p.Name);
        else
            throw new Exception("Невозможно сравнить два объекта");
    }
}
```

```
class Person : IComparable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int CompareTo(Person p)
    {
        return this.Name.CompareTo(p.Name);
    }
}
```

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };
Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people);
foreach(Person p in people) { Console.WriteLine("{0} - {1}", p.Name, p.Age);}
```

# Интерфейс IComparer

```
interface IComparer
{
    int Compare(object o1, object o2);
}
class PeopleComparer : IComparer<Person>
{
    public int Compare(Person p1, Person p2)
    {
        if (p1.Name.Length > p2.Name.Length)    return 1;
        else if (p1.Name.Length < p2.Name.Length)    return -1;
        else    return 0;}}}
```

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };
```

```
Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people, new PeopleComparer());
```

```
foreach(Person p in people) Console.WriteLine("{0} - {1}", p.Name, p.Age);
```

# Пример реализации интерфейса

```
class Monster : IComparable
```

```
{ public int CompareTo( object obj )      // реализация интерфейса
    { Monster temp = (Monster) obj;
      if ( this.health > temp.health ) return 1;
      if ( this.health < temp.health ) return -1;
      return 0; }
... }
```

```
class Class1
```

```
{ static void Main()
```

```
{ const int n = 3;
```

```
Monster[] stado = new Monster[n];
```

```
stado[0] = new Monster( 50, 50, "Вася" );
```

```
stado[1] = new Monster( 80, 80, "Петя" );
```

```
stado[2] = new Monster( 40, 10, "Маша" );
```

```
Array.Sort( stado );      // сортировка стала возможной
```

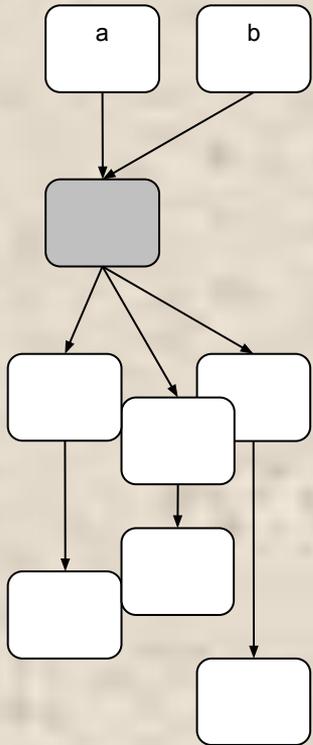
```
}}
```

# Параметризованные интерфейсы

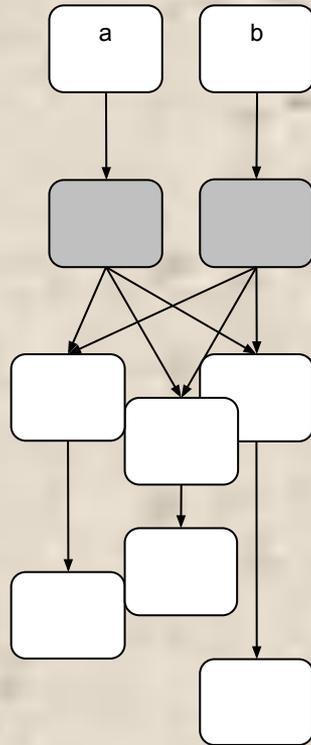
```
class Program {  
    class Elem : IComparable<Elem>  
    { string data;  
      int key;  
      ...  
      public int CompareTo( Elem obj )  
      { return key - obj.key; }  
    }  
    static void Main(string[] args)  
    {  
        List<Elem> list = new List<Elem>();  
        for ( int i = 0; i < 10; ++i ) list.Add( new Elem() );  
        list.Sort();  
        ...  
    }  
}
```

# Клонирование объектов

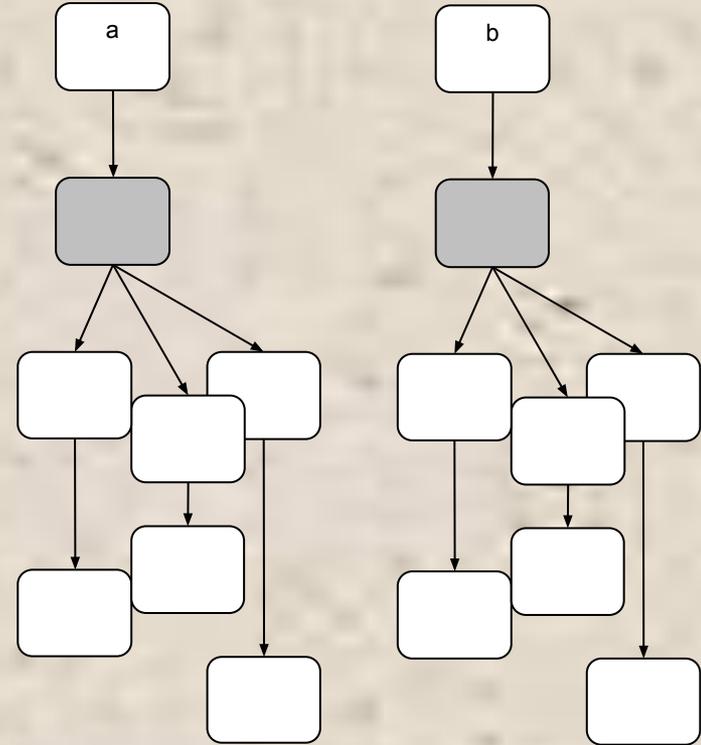
- *Клонирование* — создание копии объекта. Копия объекта называется клоном.



a)  
присваивание  
 $b = a$



б) поверхностное  
клонирование



в) глубокое  
клонирование

# Виды клонирования

- При присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект (рис. а).
- Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом `MemberwiseClone`, который объект наследует от класса `object`. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются (рис. б). Это называется *поверхностным клонированием*.
- Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов (рис. в).
- Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.
- Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса **`ICloneable`** и переопределять его единственный метод **`Clone`**.

# Клонирование объектов. Интерфейс ICloneable

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name="Tom", Age = 23 };
        Person p2 = p1;
        p2.Name = "Alice";
        Console.WriteLine(p1.Name); // Alice

        Console.Read();
    }
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

# Интерфейс ICloneable

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public object Clone()
    {
        return new Person { Name = this.Name, Age = this.Age };
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name="Tom", Age = 23 };
        Person p2 = (Person)p1.Clone();
        p2.Name = "Alice";
        Console.WriteLine(p1.Name); // Tom

        Console.Read();
    }
}
```

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

# Поверхностное клонирование

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
```

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Company Work { get; set; }
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
class Company
{
    public string Name { get; set; }
}
```

```
Person p1 = new Person { Name="Tom", Age = 23, Work= new Company { Name =
"Microsoft" } };
Person p2 = (Person)p1.Clone();
p2.Work.Name = "Google";
p2.Name = "Alice";
Console.WriteLine(p1.Name); // Tom
Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft
```

# Глубокое клонирование

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Company Work { get; set; }

    public object Clone()
    {
        Company company = new Company { Name = this.Work.Name };
        return new Person
        {
            Name = this.Name,
            Age = this.Age,
            Work = company
        };
    }
}

class Company
{
    public string Name { get; set; }
}
```