

## Интерфейсы

*Интерфейс* — именованный набор сигнатур методов. Все члены интерфейса неявно абстрактны (т. е. не содержат реализации и по умолчанию являются открытыми).

### Интерфейс `IDisposable` и освобождение ресурсов

```
public interface IDisposable
{
    void Dispose();
}
```

Пример:

```
object fs = new System.IO.FileStream(@"c:\a.txt",
    System.IO.FileMode.Create);
if (fs is IDisposable)
{
    IDisposable d = (IDisposable)fs;
    d.Dispose();
}
```

Более эффективная реализация:

```
object fs = new System.IO.FileStream(@"c:\a.txt",
    System.IO.FileMode.Create);
IDisposable d = fs as IDisposable;
if (d != null)
    d.Dispose();
```

Рекомендации, связанные с семантикой удаления объекта:

- однажды удаленный объект восстановлению не подлежит;
- повторный вызов метода `Dispose` не приводит к ошибке;
- если удаляемый объект так или иначе «владеет» другими объектами с интерфейсом `IDisposable`, то при своем удалении он должен удалить свои подчиненные объекты.

Примеры объектов, связанных с неуправляемыми ресурсами, но не требующих вызова `Dispose`: `Brushes.Blue`, `BinaryReader` (если к файловому потоку подключен и объект `BinaryWriter`).

### Оператор `using`

```
using
    (var fs = new System.IO.FileStream(@"c:\a.txt",
        System.IO.FileMode.Create))
    fs.Write(new byte[]{0, 0, 0}, 0, 3);
```

Преобразуется в:

```
var fs = new System.IO.FileStream(@"c:\a.txt",
    System.IO.FileMode.Create);
try
{
    fs.Write(new byte[]{0, 0, 0}, 0, 3);
}
finally
{
    if (fs != null)
        fs.Dispose();
}
```

### Сборщик мусора и финализаторы

Метод `Dispose` освобождает *неуправляемые ресурсы*. Память, выделенная для размещения объекта в куче, освобождается *сборщиком мусора* (garbage collector, GC). С классом может быть связан особый метод — *финализатор*, который выполняется непосредственно перед тем моментом, когда сборщик мусора разрушает объект.

Стандартная схема реализации освобождения неуправляемых ресурсов в финализаторе:

```
class DisposeDemo: IDisposable
{
    public void Dispose()
```

```
// не требуется переопределять для потомков
{
    Dispose(true);
    GC.SuppressFinalize(this);
    // подавление вызова финализатора при разрушении
}
protected virtual void Dispose
    (bool disposeIsCalled) // можно переопределять
{
    if (disposeIsCalled)
    {
        // освобождение ресурсов подчиненных объектов
    }
    // освобождение собственных ресурсов
}
~DisposeDemo()
{
    Dispose(false);
}
}
```

Схема работы сборщика мусора:

- сборщик мусора начинает процесс сборки, если объем объектов, созданных после последней сборки мусора, превысил некоторое пороговое значение (обычно 256 К);
- сборщик мусора приостанавливает выполнение приложения, начинает перебирать корневые ссылки на объекты и обходит граф объектов, помечая объекты, которых он коснется, как доступные. По окончании этого процесса все непомеченные объекты считаются мусором;
- объекты, объявленные мусором, и не имеющие финализаторов, уничтожаются немедленно. Если же они имеют финализаторы, то они помечаются как подлежащие дополнительной обработке, и для них финализаторы вызываются в отдельной нити (происходит как бы их «воскрешение»);
- все оставшиеся объекты (в том числе и те, для которых были вызваны финализаторы) перемещаются вниз по куче (происходит уплотнение кучи). Все выжившие объекты переходят в следующее поколение.

Для оптимизации сборки мусора все объекты разделяются на три поколения: только что созданные (поколение 0), объекты, пережившие одну сборку мусора (поколение 1), и объекты, пережившие хотя бы одну сборку мусора в поколении 1 («долгожители» — поколение 2). Если размер объектов из поколения 0 превышает 256 К, то они подвергаются сборке мусора; «выжившие» помещаются в поколение 1. Если при сборке мусора выясняется, что поколение 1 занимает более 2 М, выполняется сборка мусора в поколении 1, а «выжившие» при этом объекты переходят в поколение 2. Поколение 2 подвергается сборке мусора только если его размер превысит 10 М.

Принудительная сборка мусора: `GC.Collect()` или `GC.Collect(n)` для сборки мусора в поколениях, номер которых не превышает `n`.

Гарантированное уничтожение объектов с финализаторами:

```
GC.Collect();
GC.WaitForPendingFinalizers(); // ждать, пока
// не завершится нить с запущенными финализаторами
GC.Collect();
```

### Интерфейс `ICloneable` и явная реализация интерфейсов

Реализация метода `Dispose` — пример *неявной* реализации интерфейса.

При *явной* реализации интерфейса перед именем интерфейсных методов указывается имя интерфейса; в этом случае эти методы могут быть вызваны только из интерфейсных объектов.

Пример: интерфейс ICloneable:

```
public interface ICloneable
{
    object Clone(); // возвращает копию (клон) объекта
}
```

Реализация:

```
class Demo: ICloneable
{
    public int X, Y;
    public object Clone()
    {
        return new Demo { X = this.X, Y = this.Y };
    }
}
```

Пример его применения:

```
Demo a = new Demo { X = 10, Y = 20 };
Demo b = (Demo)a.Clone();
// требуется явное приведение типа - неудобно
Console.WriteLine(b.X+ " " +b.Y); // 10 20
```

Явная реализация интерфейса ICloneable:

```
class Demo: ICloneable
{
    public int X, Y;
    public Demo Clone()
    {
        return new Demo { X = this.X, Y = this.Y };
    }
    object ICloneable.Clone()
    // атрибут public указывать нельзя
    {
        return Clone(); // вызывается метод Clone класса
    }
}
```

Атрибут виртуальности при явной реализации не нужен, так как при вызове метода с помощью интерфейсного объекта позднее связывание реализуется автоматически (т. е. объект и так ведет себя как виртуальный).

Варианты вызова методов Clone:

```
Demo a = new Demo { X = 10, Y = 20 };
Demo b = a.Clone(); // вызывается метод Clone класса
Console.WriteLine(b.X + " " + b.Y); // 10 20
Demo c = (Demo)(a as ICloneable).Clone();
// вызывается метод Clone интерфейса
Console.WriteLine(c.X + " " + c.Y); // 10 20
```

### Интерфейсные методы и позднее связывание

При вызове метода посредством интерфейсного объекта он всегда ведет себя как *виртуальный* (даже если он реализован неявно и при этом не имеет атрибута виртуальности). Однако при вызове этого же метода непосредственно из классового объекта позднее связывание будет доступно только при включении этого метода в цепочку виртуальности. **Пример:**

```
using System;
public interface ICommon
{
    void DoIt();
}
public class Base: ICommon
{
    void ICommon.DoIt() { Console.Write("ICommon(Base) "); }
    public virtual void DoIt() { Console.Write("Base "); }
}
public class Derived1: Base, ICommon
```

```
{
    void ICommon.DoIt()
    { Console.Write("ICommon(Derived1) "); }
    public new virtual void DoIt()
    { Console.Write("Derived1 "); }
}
public class Derived2: Derived1
{
    public override void DoIt()
    { Console.Write("Derived2 "); }
}
class Program
{
    static void Main()
    {
        Derived2 r1 = new Derived2();
        Derived1 r2 = r1;
        Base r3 = r1;
        ICommon r4 = r1;
        r1.DoIt(); r2.DoIt(); r3.DoIt(); r4.DoIt();
        Console.ReadLine();
    }
}
```

Результат работы программы:

```
Derived2 Derived2 Base ICommon(Derived1)
```

Если в описании класса Derived1 убрать реализацию метода ICommon.DoIt, то четвертый вызов выведет «Derived2».

Если дополнительно убрать явное подключение интерфейса ICommon к классу Derived1, то четвертый вызов выведет «ICommon(Base)».

Заменяем атрибуты new virtual в методе DoIt класса Derived1 на атрибут override:

```
Derived2 Derived2 Derived2 ICommon(Base)
```

Если после всех описанных изменений убрать реализацию метода ICommon.DoIt в описании класса Base, то все четыре вызова методов выведут «Derived2».

Убрав дополнительно все атрибуты virtual и override, мы получим следующий результат:

```
Derived2 Derived1 Base Base
```

Добавив к описанию класса Derived2 явное подключение интерфейса ICommon, получим:

```
Derived2 Derived1 Base Derived2
```

Если теперь убрать реализацию метода DoIt в классе Derived2, то получим следующий результат:

```
Derived1 Derived1 Base Base
```

Если добавить к классу Derived1 интерфейс ICommon, то последний вызов выведет «Derived1».

### Базовые интерфейсы для коллекций и цикл foreach

Под *коллекциями* в широком смысле подразумеваются классы, содержащие набор элементов. Интерфейс *перечислителя*:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Для получения перечислителя в классе-коллекции предусматривается специальный метод GetEnumerator:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Для того чтобы некоторый объект допускал перебор своих элементов, достаточно, чтобы он реализовывал интерфейс `IEnumerable`. Пример (предполагается, что `s` — коллекция элементов `char`; она может, например, иметь тип `string` или `char[]`):

```
IEnumerator en = s.GetEnumerator();
while (en.MoveNext())
{
    char c = (char)en.Current;
    Console.WriteLine(c);
}
```

Цикл `foreach` упрощает перебор:

```
foreach (char c in s)
    Console.WriteLine(c);
```

### Итераторы и конструкция `yield`

Тогда как цикл `foreach` упрощает использование коллекций, итераторы (конструкция `yield`) упрощают ее создание.

```
class YieldDemo: IEnumerable
{
    public string[] S = new string[10];
    public IEnumerator GetEnumerator()
    {
        foreach (string s in S)
            if (s != null)
                foreach(char c in s)
                    yield return c;
    }
}
static void Main(string[] args)
{
    YieldDemo d = new YieldDemo();
    d.S[0] = "000"; d.S[5] = "555";
    foreach (char c in d)
        Console.WriteLine(c); // 0 0 0 5 5 5
    d.S[3] = "33"; d.S[5] = "55";
    foreach (char c in d)
        Console.WriteLine(c); // 0 0 0 3 3 5 5
}
```

Метод, использующий `yield`, может иметь возвращаемый тип `IEnumerator`. В этом случае его возвращаемое значение считается коллекцией с присоединенным перечислителем:

```
static IEnumerable Fib(int n)
{
    int a = 1, b = 1, c;
    yield return a;
    yield return b;
    for (int i = 2; i < n; i++)
    {
        c = a + b;
        yield return c;
        a = b; b = c;
    }
}
static void Main(string[] args)
{
    foreach (int f in Fib(10))
        Console.WriteLine(f); // 1 1 2 3 5 8 13 21 34 55
}
```

После завершения перебора элементов перечислитель, созданный с помощью оператора `yield`, не может быть использован повторно. Если требуется досрочно прервать перебор элементов, то надо использовать оператор `yield break`:

```
static IEnumerable TwoOrThree(bool onlyTwo)
{
    yield return 1; yield return 2;
    if (onlyTwo)
        yield break;
    yield return 3;
}
```

### Интерфейс `IComparable` и сравнение объектов

```
Array.Sort(a);
```

Элементы должны реализовывать интерфейс `IComparable`:

```
public interface IComparable
{
    int CompareTo(object other);
}
```

Все числовые типы (а также типы `bool`, `char` и `string`) реализуют интерфейс `IComparable`.

```
Point[] p = new Point[5];
```

```
Array.Sort(p); // исключение InvalidOperationException
```

Можно определить «пользовательский» алгоритм сравнения, оформив его в виде класса-компаратора:

```
public interface IComparer
{
    int Compare(object a, object b);
}
```

Пример:

```
using System;
using System.Drawing;
using System.Collections;
```

```
class PointComparer: IComparer
{
    public int Compare(object a, object b)
    {
        Point pa = (Point)a, pb = (Point)b;
        long da = (long)pa.X * pa.X + (long)pa.Y * pa.Y,
            db = (long)pb.X * pb.X + (long)pb.Y * pb.Y;
        return Math.Sign(da - db);
    }
}
```

```
...
static void Main(string[] args)
```

```
{
    Point[] p = new Point[4];
    p[0].X = int.MaxValue;
    p[2].Y = 100;
    p[3].X = 40;
    Array.Sort(p, new PointComparer());
    foreach (Point a in p)
        Console.Write(a);
}
```

Результат:

```
{X=0,Y=0} {X=40,Y=0} {X=0,Y=100} {X=2147483647,Y=0}
```

Интерфейс, связанный со сравнением объектов на равенство:

```
public interface IEqualityComparer
{
    bool Equals(object a, object b);
    int GetHashCode(object obj);
}
```

Методы этого интерфейса должны быть согласованы:

- равные объекты должны иметь одинаковые хеш-коды;
- значения хеш-кодов должны быть по возможности равномерно распределены по всему целочисленному диапазону.