Интерфейсы

Интерфейсом называется семейство методов и свойств, которые сгруппированы в единое целое и инкапсулируют какую-либо определенную функциональную возможность

После того как интерфейс определен, его можно реализовать в некотором классе. Это означает, что класс будет поддерживать все свойства и члены, определяемые данным интерфейсом.

Интерфейсы не могут существовать сами по себе.

Интерфейс представляет собой не более чем просто именованный набор абстрактных членов. В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации.

В нем указывается только, что именно следует делать, но не как это делать.

Интерфейс не может содержать в себе никакого кода, который бы реализовал его члены; он просто *описывает эти члены*. Их реализация должна находиться в классах, в которых реализован данный интерфейс.

У класса может быть несколько интерфейсов, и несколько классов могут поддерживать один и тот же интерфейс. С помощью интерфейсов можно включить поведение из нескольких источников в классе.

Эта возможность важна в C#, поскольку язык не поддерживает множественное наследование классов.

В C++ была возможность множественного наследования. Разработчики C# решили отказаться от этого и придумали интерфейсы.

Класс не может быть унаследован от нескольких классов, но при этом унаследовать несколько интерфейсов.

Интерфейсы нужны для наследования. Часто необходимо реализовать несколько классов, при этом у них одинаковые методы (название), но по разному должны быть реализованы.

К примеру можно создать несколько классов: круг, квадрат, треугольник.

У всех классов необходимо реализовать методы вычисления площади и периметра.

Для этого нужно создать интерфейс с двумя методами. После каждый класс унаследует этот интерфейс, и по своему реализует методы.

Интерфейсы объявляются похожим на классы способом, только вместо ключевого слова class используется ключевое слово interface. Например:

```
interface IMyInterface
    {
        // члены интерфейса
    }
```

Ключевые слова для модификации доступа **public** и **internal** используются точно так же. Для того, чтобы сделать интерфейс общедоступным, следует использовать ключевое слово **public**:

```
public interface IMyInterface
    {
        // члены интерфейса
    }
```

Фактически это те же самые *абстрактные классы*, не содержащие объявлений данных-членов и объявлений обычных функций. Все без исключения функции — члены интерфейса – *абстрактные*.

Пусть определен интерфейс

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Любой класс или структура, реализующие интерфейс IEquatable<T>, должны содержать определение метода Equals, который соответствует сигнатуре, при определении интерфейса.

Методы, реализующие интерфейс, должны быть объявлены как public. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой.

Интерфейс определяет только сигнатуру.

Интерфейс может быть членом пространства имен или класса и содержать сигнатуры следующих членов:

- **методы,**
- **свойства,**
- **события,**
- **индексаторы**

Для реализации члена интерфейса соответствующий член класса должен быть открытым, нестатическим, и иметь то же имя и сигнатуру, что и член интерфейса.

- Интерфейс не может содержать **константы, поля, операторы, конструкторы, деструкторы экземпляра, или типы**.
- Члены интерфейса автоматически открыты, и они не могут включать модификаторы доступа.
- Члены также не могут быть статическими.

```
public class Car : IEquatable<Car>
 {
   public string Make  { get; set;}
   public string Model { get; set; }
   public string Year  { get; set; }

   // Implementation of IEquatable<T> interface

   public bool Equals(Car car)
     {
         if (this.Make == car.Make &&
             this.Model == car.Model && this.Year == car.Year)
           { return true; }
          else
              return false;
     }
}
```

Свойства можно объявлять в **interface** .

Ниже приведен пример метода доступа **индексатора** интерфейса:

```
public interface ISampleInterface
{
   string Name { get; set; }
}
```

Метод доступа свойства интерфейса не имеет тела. Поэтому методы доступа предназначены для того, чтобы указывать, доступно ли свойство для чтения и записи, только для чтения или только для записи.

Можно использовать полное имя свойства, на которое ссылается на интерфейс, в классе.

```
interface IEmployee
 {
   string Name { get; set; }
   int Counter { get; }
 }
```

```
public class Employee : IEmployee  // Интерфейс задает поведение
{                                  // name и counter
  public static int numberOfEmployees;
  private string name;

  public string Name // read-write
   {
     get { return name; }
     set { name = value; }
    }
   private int counter;
   public int Counter // read-only
   {
      get { return counter; }
   }

   public Employee() // Конструктор класса
    {
       counter = ++counter + numberOfEmployees;
    }
}
```

```
class TestEmployee
{
 static void Main()
  {
    System.Console.Write("Enter number of employees: ");
    Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

    Employee e1 = new Employee();
    System.Console.Write("Enter the name of the new employee: ");
    e1.Name = System.Console.ReadLine();

    System.Console.WriteLine("The employee information:");
    System.Console.WriteLine("Employee number: {0}", e1.Counter);
    System.Console.WriteLine("Employee name: {0}", e1.Name);
  }
}
```

Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous

The employee information:
Employee number: 211
Employee name: Hazem Abolrous

Наследование для интерфейсов определяется аналогично наследованию для классов. Основное отличие здесь в том, что мы можем использовать интерфейсы с множественными базами, например:

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
    {
                    //
    }
```

Интерфейсы, как и классы, наследуются от **System.object**. Этот механизм допускает полиморфизм интерфейсов. Однако, как отмечалось ранее, нельзя создать экземпляр интерфейса таким же способом, как и экземпляр класса.

Для интерфейсов ключевые слова **abstract** и **sealed** использовать нельзя, так как ни один модификатор для интерфейсов не имеет смысла (у них отсутствует реализация, следовательно, для них не могут создаваться экземпляры в явном виде)

Если класс реализует два интерфейса, содержащих член с одинаковой сигнатурой, то при реализации этого члена в классе оба интерфейса будут использовать этот член для своей реализации.

В следующем примере все вызовы **Paint** вызывают один метод.

```
interface Icontrol  {    void Paint();   }
interface Isurface  {    void Paint();   }

class SampleClass : IControl, Isurface
 {
    public void Paint()
     {
        Console.WriteLine("Paint method in SampleClass");
     }
}
```

```
class Test
{
  static void Main()
   {
     SampleClass sc = new SampleClass();
     IControl ctrl = (IControl)sc;
     ISurface srfc = (ISurface)sc;

     sc.Paint();                    // Вызывается один метод
     ctrl.Paint();
     srfc.Paint();
   }
}


// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

```
interface Icontrol  {    void Paint();   }
interface Isurface  {    void Paint();   }

class SampleClass : IControl, Isurface
 {
    public void Paint()
     {
        Console.WriteLine("Paint method in SampleClass");
     }
}
```

Возможна и явная реализация члена интерфейса — путем создания члена класса, который вызывается только через интерфейс и имеет отношение только к этому интерфейсу. Это достигается путем включения в имя члена класса имени интерфейса с точкой.

```
public class SampleClass : IControl, Isurface
 {
   void IControl.Paint()
    {
      System.Console.WriteLine("IControl.Paint");
    }

    void ISurface.Paint()
    {
      System.Console.WriteLine("ISurface.Paint");
    }
}
```

Явная реализация также используется для разрешения случаев, когда каждый из двух интерфейсов объявляет разные члены с одинаковым именем, например свойство и метод.

```
interface ILeft
{
  int P { get;} // Свойство
}
interface IRight
{
   int P();     // Метод
}


class Middle : ILeft, Iright
 {
   public int P() { return 0; }
   int ILeft.P    { get { return 0; } }
 }
```

Интерфейс IDisposable. *Освобождение ресурсов*

Одним из интерфейсов, представляющих особенный интерес, является интерфейс iDisposable.

Основное назначение этого интерфейса заключается в высвобождении неуправляемых ресурсов. Сборщик мусора автоматически высвобождает память, выделенную для управляемого объекта, если этот объект уже не используется. Однако он не может предсказать момент выполнения сбора мусора. Кроме того, у сборщика мусора нет сведений о неуправляемых ресурсах, таких как дескрипторы окон, или открытые файлы и потоки.

Метод Dispose( ) этого интерфейса используется вместе со сборщиком мусора для высвобождения неуправляемых ресурсов явным образом. Пользователь объекта может вызвать этот метод, когда объект ему больше не нужен.

На объекте, который поддерживает интерфейс iDisposabie, должен быть реализован метод Dispose( ), т.е. в нем должен содержаться код для реализации этого метода.

Этот метод может вызываться, когда объект больше не нужен (например, перед выходом из области его действия); он освобождает некие критические ресурсы, которые в противном случае будут удерживаться до тех пор, пока сборкой мусора не будет запущен метод деструктора. Это дает дополнительные возможности для управления ресурсами, которые используются объектами.

Совмещение освобождения ресурсов с удалением объекта-владельца данного ресурса называется *недетерминированным освобождением*. Зависящее от деятельности сборщика мусора *недетерминированное* освобождение ресурсов **не всегда является оптимальным решением**:

- время начала очередного цикла работы GC обычно неизвестно;
- будет ли удален соответствующий объект в случае незамедлительной активизации GC;
- что делать, если ресурс необходимо освободить незамедлительно, а объект должен быть сохранен.

Для *детерминированного* освобождения неуправляемых ресурсов может быть использован какой-либо специальный метод. Этот метод желательно сделать универсальным, по крайней мере, в смысле его объявления и вызова

В этом случае для любого вновь объявляемого класса можно предположить существование метода с предопределенным именем и сигнатурой (спецификацией типа возвращаемого значения и списком параметров), который можно было бы стандартным образом вызывать непосредственно от имени объекта-владельца ресурса вне зависимости от активности системы очистки памяти.

В .NET с этой целью используется интерфейс IDisposable с методом Dispose().

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

Таким образом, для реализации механизма освобождения ресурсов достаточно наследовать интерфейс IDisposable и обеспечить реализацию метода Dispose.

После этого деятельность по освобождению ресурсов сводится к теперь стандартному вызову метода Dispose

Следующие правила определяют в общих чертах рекомендации по использованию метода Dispose.

- В методе **Dispose** освобождаются любые ресурсы, которыми владеет объект данного типа и которые можно освободить.
- Если для освобождения ресурсов, которыми владеет объект, не был вызван метод **Dispose**, неуправляемые ресурсы должны освобождаться в методе **Finalize**.
- Метод **Dispose** может дублировать действия по освобождению ресурсов, предпринимаемые в деструкторе (**Finalize**) при уничтожении объекта. Для этого можно предусмотреть систему булевых переменных, связанных с состоянием ресурса, или вызов для данного объекта метода **GC.SuppressFinalize**, который для этого объекта запретит выполнение кода деструктора, соответствующего в C# коду метода **Finalize**. Означает ли это, что GC уничтожит объект, не передавая при этом управление деструктору?
- Метод **Dispose** должен освобождать все ресурсы, удерживаемые данным объектом и любым объектом, которым владеет данный объект.
- Следует обеспечить возможность многократного вызова метода. При этом желательно обойтись без генерации исключений. **Dispose** просто не должен пытаться повторно освобождать ранее освобожденные ресурсы.

```
using System;

namespace TestD00
{
   public class MemElem : IDisposable
    {
       bool dFlag;
       int i;
       public MemElem(int iKey) { dFlag = true; i = iKey; }

       ~MemElem()                                    // Деструктор (он же Finalizer)
          {   Console.WriteLine("{0} disposed : {1}!", i, (dFlag == true) ? "yes" : "no");
              if (!dFlag)
                    Dispose( );                  // Если ресурсы не освобождены – вызывается Dispose.
          }

       public void Dispose()   // Детерминированное управление ресурсом: имитация активности.
          {
               if (dFlag==true)  { Console.WriteLine("Dispose is here!"); dFlag = false; }
                  else           { Console.WriteLine("Dispose was here!"); }
           }
      }
```

```
class Program
{
  static void Main(string[ ] args)
    {
      IDisposable d;
      MemElem mmm = null;
      int i;
      for (i = 0; i < 10; i++)
          {
            MemElem m = new MemElem(i);
            if (i == 5)
              {
                d = m as IDisposable;
                if (d != null) d.Dispose();
                mmm = m;
                GC.SuppressFinalize(mmm);  // И закрыли для кода деструктора.
               }
          }
```

```
      Console.WriteLine("-----0-----");
       GC.Collect();  Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
    // Заменить false на true. Осознать разницу.
       Console.WriteLine("-----1-----");
     // А теперь вновь разрешили его удалить... Пока существует хотя бы одна ссылка – GC
     //  все равно не будет уничтожать этот объект.

       if (mmm != null) GC.ReRegisterForFinalize(mmm);  mmm = null;

  // Вот потеряли ссылку на объект. Теперь GC может уничтожить объект. Если разрешена
  // финализация - будет выполнен код деструктора. Возможно, что отработает метод Dispose.
  // Если финализация запрещена - код деструктора останется невыполненным.

       Console.WriteLine("-----2-----");
       GC.Collect();

    // Заменить false на true. Осознать разницу.
       Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
       Console.WriteLine("-----3-----");
}
}
}
```

Документация Microsoft о применении IDisposable довольно запутанная. На самом деле она упрощается до трех простых правил.

- Правило первое: не применять (до тех пор, пока это действительно не понадобится)
- Правило второе: для класса, владеющего управляемыми ресурсами, реализуйте Disposable (но не финализатор)
- Правило третье: для класса, владеющего неуправляемыми ресурсами, реализуйте IDisposable и финализатор

Оператор using

Оператор using предоставляет удобный синтаксис, обеспечивающий правильное использование объектов IDisposable.