№ 1 -2 Основы CLR и .NET. Проектирование типов

Задание

1) Подготовить ответы на все вопросы.
    1. Что такое .Net Framework и из чего он состоит?
    2. Поясните, что такое CLR-среда.
    3. Что такое FCL?
    4. Какая наименьшая исполнимая единица в .NET?
    5. Что такое IL?
    6. Пояснить работу JIT-компилятора?
    7. Что такое CTS (Common Type System)?
    8. Какие аспекты поведения определяет тип System.Object?
    9. Что находится в MSCorLib dll?
    10. Что такое частные и общие сборки?
    11. Что такое assembly manifest?
    12. Что такое GAC?
    13. Чем managed code отличается от unmanaged code
    14. Как используется ключевое слово unsafe в C#?
    15. Как и для чего определен метод Main?
    16. Поясните состав и назначение config файла.

2)

Определить класс, содержащий:

- конструкторы (с параметрами и без);
- **статический конструктор** (конструктор типа);
- поле - **только для чтения** (например, для каждого экземпляра сделайте поле только для чтения ID - равно некоторому уникальному номеру (хэшу) вычисляемому автоматически на основе инициализаторов объекта);
- константу;
- статическое поле– количество объектов;
- **свойства** (get, set) – для всех данных членов класса;
- в одном из методов класса для работы с аргументами используйте **ref - и out-параметры**.
- создайте в классе **статический метод** вывода информации о классе.
- переопределяете методы Equals и GetHashCode (унаследованы от Object) для сравнения объектов; для алгоритма вычисления хэша руководствуйтесь стандартными рекомендациями.

Создайте статический класс (например MathObject) содержащий методы математического преобразования над объектом вашего класса

или расчета определенных параметров (например: уменьшение, поворот, площадь, периметр и т.п). Позже добавьте к нему метод расширения – например проверки возможности упаковки вашей геометрической фигуры в коробку размера a,b,c).

Создайте и выведите анонимный тип (по образцу вашего класса).

3) Ответьте на вопросы, приведенные ниже

Используйте документацию http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx

Далее приведен перечень классов:

Вариант	Задание
1	Классы – фигура (набор точек)
2	Классы – куб;
3	Классы – пирамида;
4	Классы – ромб;
5	Классы – цилиндр;
6	Классы – тор;
7	Классы – призма;
8	Классы – конус
9	Классы – усеченный конус;
10	Классы – многогранник;
11	Классы – параллелепипед;
12	Классы – прямая
13	Классы – треугольная призма;
14	Классы – звезда;
15	Классы – снежинка;

Вопросы

1. Назовите класс .NET, от которого наследуются все классы.

2. Охарактеризуйте открытые методы System.Object.
3. Охарактеризуйте закрытые методы System.Object.
4. Варианты использования директивы using(using Directive) в C#.
5. Как связаны между собой сборки и пространства имен?
6. Что такое примитивные типы данных?
7. Что такое ссылочные типы? Какие типы относятся к ним?
8. Какие типы относятся к типам-значениям?
9. В чем отличие между ссылочными и значимыми типами данных?
10. Как конвертировать значимый тип в ссылочный?
11. Что происходит в памяти при упаковке и распаковке значимого типа?
12. Что будет выведено в результате выполнения

```
class A
        {
            private int _num;
            public A(int num) { Num = num; }
            public int Num { get { return _num; } set { _num = value; } }
        }

        static void Main(string[] args)
        {
            A a = new A(5);
            A b = a;
            Console.WriteLine(a.Num + " "  +b.Num);

            a.Num = 7;
            Console.WriteLine(a.Num + " " + b.Num);
        }
```

13. В чем различие между классом и структурой?
14. Какие спецификаторы доступа существуют в C#?
15. Опишите модификатор protected internal.
16. Поясните явные преобразования переменных с помощью команд Convert.
17. Как выполнить консольный ввод/вывод?
18. Приведите примеры определения и инициализации одномерных и двумерных массивов.
19. Какие типы можно использовать в foreach? Приведите пример.
20. Какие есть способы для задания и инициализации строк?
21. Зачем и как используются ref и out параметры функции?
22. Приведите пример определения класса.
23. Что такое внутренние, абстрактные и герметизированные классы?
24. Приведите пример полей класса – статические, константные, только для чтения.
25. Для чего с методами класса используют - virtual , abstract, override , extern?
26. Приведите пример определения свойств класса.

27. Назовите явное имя параметра, передаваемого в метод set свойства класса?
28. Что такое автоматические свойства?
29. Что такое индексаторы класса?
30. Чем перекрытый метод отличается от перегруженного метода?
31. Можно ли объявить перекрытый метод статическим, если перекрываемый метод не является статическим?
32. Что такое partial класс и какие его преимущества?
33. Что такое анонимный тип в C#?
34. Для чего делают статические классы?
35. Какая разница между поверхностным (shallow) и глубоким (deep) копированием?
36. В чем разница между равенством и тождеством объектов?
37. Приведите пример метода расширения.
38. Что такое неявно типизированная переменная?

Краткие теоретические сведения

Приведенные здесь и далее теоретические сведения не являются достаточными для освоения тем (это краткий вводный материал). Необходимо использовать дополнительную литературу!!!!!

Язык программирования C# является прямым наследником языка C++. Он унаследовал многие синтаксические конструкции языка C и объектно-ориентированную модель C++. В отличие от C++ C# является чисто объектно-ориентированным языком. В объектно-ориентированном программировании ход выполнения программы определяется объектами. Объекты это экземпляры класса. Класс это абстрактный тип данных, определяемый пользователем (программистом). Класс включает в себя данные и функции для обработки этих данных. В C# запрещены глобальные функции. Все функции должны быть обязательно определены внутри класса. Не является исключением и главная функция языка C# Main( ) (в отличии от языка C пишется с прописной буквы).

Объявление класса синтаксически имеет следующий вид:

```
class имя_класса
      {
            // члены класса
       }
```

Члены класса это данные и функции для работы с этими данными. Рассмотрим шаблон приложения, подготовленный для нас мастером:

```
using System;
namespace ConsoleApplication10
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      class Class1
```

```
        {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                //
                // TODO: Add code to start application here
                //
            }
        }
}
```

Первая строчка проекта *using System;*, включает в себя директиву *using*, которая сообщает компилятору, где он должен искать классы (типы), не определенные в данном пространстве имен. Мастер, по умолчанию, указывает стандартное пространство имен *System*, где определена большая часть типов среды .NET.

Следующей строчкой *namespace ConsoleApplication10* мастер предложения определяет пространство имен для нашего приложения. По умолчанию в качестве имени выбирается имя проекта. Область действия пространства имен определяется блоком кода, заключенного между открывающей и закрывающей фигурными скобками. Пространство имен обеспечивает способ хранения одного набора имен отдельно от другого. Имена, объявленные в одном пространстве имен не конфликтуют, при совпадении, с именами, объявленными в другом пространстве имен.

В шаблоне приложения имеется множество строк, которые являются комментариями.

В C# определены три вида комментариев:

- многострочный (/*…*/)
- однострочный (//…)
- XML (///) – комментарий для поддержки возможности создания самодокументированного кода.

Строчка [STAThread] является атрибутом. Атрибуты задаются в квадратных скобках. С помощью атрибута в программу добавляется дополнительная описательная информация, связанная с элементом кода, непосредственно перед которым задается атрибут. В нашем случае указывается однопоточная модель выполнения функции Main. Заголовок функции:

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

Функция Main определена как статическая (static) с типом возвращаемого значения void. Функция Main( ) C# как и функция main( ) языка C может принимать аргументы. Аргумент - это строковый массив, содержащий элементы командной строки. Тело функции пустое и в нем содержится, в виде комментария, предложение добавить туда код для запуска приложения:

```
// TODO: Add code to start application here
```

Воспользуемся этим предложением и добавим в тело функции одну строчку:

```
static void Main(string[] args)
{
      //
      // TODO: Add code to start application here
      Console.WriteLine("Привет!");
      //
}
```

Функции консольного ввода-вывода являются методами класса Console библиотеки классов среды .NET.

Для ввода строки с клавиатуры используется метод Console.ReadLine(), а для ввода одного символа метод Console.Read().

Для консольного вывода также имеются две метода

- метод Console.Write(), который выводит параметр, указанный в качестве аргумента этой функции, и
- метод Console.WriteLine(),который работает так же, как и Console.Write(), но добавляет символ новой строки в конец выходного текста.

Для анализа работы этих методов модифицируйте функцию Main( ) так, как показано ниже :

```
static void Main(string[] args)
{
      //
      // TODO: Add code to start application here
      Console.WriteLine("Введите ваше имя");
      string str=Console.ReadLine();
      Console.WriteLine("Привет "+str+"!!!");
      Console.WriteLine("Введите  один символ с клавитуры");
      int kod=Console.Read();
      char sim=(char)kod;
      Console.WriteLine("Код символа  "+sim+" = "+kod);

      //
}
```

Добавим

```
Console.WriteLine("Код символа  {0} = {1}",sim,kod);
```

Первым параметром списка является строка, содержащая маркеры в фигурных скобках. Маркер это номер параметра в списке. При выводе текста вместо маркеров будут подставлены соответствующие параметры из остального списка. После маркера через запятую можно указать, сколько позиций отводится для вывода значений. Например, запись {1,3} означает, что для печати первого элемента списка отводится поле шириной в три символа.

Причем, если значение ширины положительно, то производится выравнивание по правому краю поля, если отрицательно то по левому.

Добавим 4 новые строчки в конец кода функции Main():

```
int s1=255;
int s2=32;
Console.WriteLine(" \n{0,5}\n+{1,4}\n-----\n{2,5}",s1,s2,s1+s2);
Console.WriteLine(" \n{1,5}\n+{0,4}\n-----\n{2,5}",s1,s2,s1+s2);
//
```

Кроме того, после поля ширины через двоеточие можно указать форматную строку, состоящую из одного символа и необязательного значения точности.

Существует 8 различных форматов вывода:

- C – формат национальной валюты,
- D – десятичный формат,
- E – научный (экспоненциальный) формат,
- F – формат с фиксированной точкой,
- G – общий формат,
- N – числовой формат,
- P – процентный формат,
- X – шестнадцатеричный формат

Например, запись {2,9:C2} – означает, что для вывода второго элемента из списка, отводится поле шириной в 9 символов. Элемент выводится в формате денежной единицы с количеством знаков после запятой равной двум. При выводе результата происходит округление до заданной точности.

Классы, член данные и член функции класса

Класс это абстрактный тип данных, определяемый программистом (пользователем). С помощью классов определяются свойства объектов. Объекты это экземпляры класса. Объявление класса синтаксически имеет следующий вид:

```
class имя_класса
        {
                // члены класса
         }
```

Члены класса – это данные и функции для работы с этими данными.
Имя класса – это, по сути дела, имя нового типа данных.

Создание экземпляра (объекта) класса осуществляется с помощью оператора new:

Имя_класса имя_объекта = new имя_класса();

Доступ к членам класса управляем. Управление доступом осуществляется с помощью спецификаций доступа:

- public – общедоступный член класса.
- private – член класса доступен только внутри данного класса.
- protected – член класса доступен только внутри данного класса и внутри классов, производных от данного.
- internal – член класса доступен только внутри данной сборки (программы).

По умолчанию в классе устанавливается спецификация доступа private. Спецификация доступа, отличная от private, должна указываться явно перед каждым членом класса.

Данные класса подразделяются на *поля, константы и события*.
Поле – это обычная член-переменная, содержащая некоторое значение.
Можно, например, объявить класс, членами которого являются только поля:

```
class CA
{
      public           int        x;
      protected  float      z;
                       double     m;
      public           char  sim;
      private          decimal    sum;

}
```

Поле **m** здесь объявлено по умолчанию приватным.

Константы – это поле, объявленное с модификатором const, или, другими словами. Это поле, значение которого изменить нельзя, например:

```
      public     const      int x = 25;
```

Все член функции класса имеют неограниченный доступ ко всем член данным класса независимо от спецификации доступа. Член функции класса в свою очередь подразделяются на:

- методы
- свойства
- конструкторы
- деструкторы
- индексаторы
- операторы.

Методы

В основном, с помощью методов класса осуществляется обработка член данных класса. Другими словами, методы определяют поведение экземпляров данного класса. Методы класса это обычные функции С - стиля. В отличии от

функций C, при передаче методу параметров по адресу, необходимо указывать ключевое слово **ref** или **out**. Эти ключевые слова сообщают компилятору, что адреса параметров функции совпадают с адресами переменных, передаваемых в качестве параметров. Любое изменение значения параметров в этом случае приведет к изменению и переменных вызывающего кода. Рекомендуется для входного параметра использовать ключевое слово **ref,** а для выходного параметра ключевое слово **out**, так как параметр функции с ключевым словом **ref** должен быть обязательно проинициализирован перед вызовом функции. При вызове методов указание ключевых слов **ref** и **out** обязательно. Методы могут быть объявлены с ключевым словом static например:

```
public static int minabs(ref int x,ref int y)
{
      //тело функции
}
```

В этом случае для вызова метода имя класса, в котором она определена, и через точку имя метода:

```
Cmin.minabs(ref a,ref b);
```

Точка в C# означает принадлежность функции данному классу (в нашем случае Cmin).

ПРИМЕР.

```
using System;

namespace ConsoleApplication12
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      class Cmin
      {
            public static int min(int x,int y)
            {
                  int z = (x<y)?x:y;
                  return z;
            }
            public static int minabs(ref int x,ref int y)
            {
                  x = (x<0)?-x:x;
                  y = (y<0)?-y:y;
                  int z = (x<y)?x:y;
                  return z;
            }

      }

      class Class1
      {
            /// <summary>
```

```
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            int a=-4;
            int b=2;
            Console.WriteLine("a={0}  b={1}",a,b);
            int k =Cmin.min(a,b);
            Console.WriteLine("a={0}  b={1}",a,b);
            Console.WriteLine("k="+k);
            k =Cmin.minabs(ref a,ref b);
            Console.WriteLine("a={0}  b={1}",a,b);
            Console.WriteLine("k="+k);

            //
        }
    }
}
```

Свойства

Свойства в C# состоят из объявления поля и методов-аксессоров для работы с этим полем.

Эти методы- аксессоры называются получатель (get) и установщик (set). Например, простейшее свойство **y**, работающее с полем **m,** можно представить следующим образом:

```
        private int m=35;
        public  int y
        {
            get
            {
                return m;
            }
            set
            {
                m=value;
            }
        }
```

Свойство, определяется, так же как и поле, но после имени свойства идет блок кода, включающий в себя два метода get и set. Код этих методов может быть сколь угодно сложным, но в нашем случае это всего лишь один оператор. Аксессор **get** всегда возвращает значение того типа, который указан в определении свойства. Аксессор **set** всегда принимает в качестве параметра переменную **value**, которая передается ему неявно. Один из аксессоров может быть опущен, в этом случае мы получаем поле только для чтения или только для записи.

Обращение к свойству осуществляется точно так же как и к полю.

ПРИМЕР.

```
using System;

namespace ConsoleApplication11
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>

	class CStatic
	{
		private int m=35;
		public  int y
		{
			get
			{
				return m;
			}
			set
			{
				m=value;
			}
		}
	}

	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main(string[] args)
		{
			//
			// TODO: Add code to start application here
			CStatic p=new CStatic();//создается экземпляр класса
			Console.WriteLine("{0}",p.y);
			p.y=75;
			int z = p.y;
			Console.WriteLine("{0}",z);

			//
		}
	}
}
```

Индексаторы

Индексаторы позволяют приложению обращаться с объектом класса так, как будто он является массивом. Индексатор во многом напоминает свойство, но в отличие от свойства он принимает в качестве параметра индекс массива. Так как объект класса используется как массив, то в качестве имени класса

используется ключевое слово **this**. Определение индексатора синтаксически выглядит следующим образом:

```
public float this[int j]
	{
		get
		{
			//Возврат необходимых данных
		}
		set
		{
			//Установка необходимых данных
		}
	}
```

Пример приложения, использующего индексаторы, приведен ниже:

ПРИМЕР

```
using System;

namespace ConsoleApplication13
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	class Rmas
	{
		protected float[] msf=new float[10];
		public float this[int j]
		{
			get
			{
				return msf[j];
			}
			set
			{
				msf[j]=value;
			}
		}

	}
	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main(string[] args)
		{
			Rmas obj = new Rmas();
			for(int i=0; i<10;i++)
			{
				obj[i] = (float)1.5*i;
			}
			for(int i=0; i<10;i++)
```

```
                {
                        Console.WriteLine("{0}",obj[i]);
                }
        }
    }
}
```

Конструкторы, поля только для чтения, вызов конструкторов.

Конструктор – это метод класса, имеющий имя класса. Конструкторов в классе может быть несколько или ни одного.

На конструкторы накладываются следующие ограничения:

1. Конструктор не может иметь возвращаемого значения даже **void**
2. Как следствие 1 нельзя использовать оператор **return()**
3. Конструкторы нельзя объявлять виртуальными.

Конструктор автоматически вызывается на этапе компиляции при создании экземпляра данного класса. Попытка вызвать конструктор явным образом вызовет ошибку компиляции.

Различают следующие типы конструкторов:

1. Конструктор по умолчанию
2. Конструктор с аргументами

Конструктор по умолчанию

Конструктор, объявленный без аргументов, называется конструктором по умолчанию.

Если в классе не определен ни один конструктор, то компилятор сам автоматически создает конструктор по умолчанию, который инициализирует все поля класса своими значениями по умолчанию (для числовых значений ноль, для булевской переменной FALSE, для строк пустые ссылки).

Пример:

// В классе CA нет явно объявленных конструкторов

```
using System;

namespace ConsoleApplication15
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>


    class CA
    {
        public int x,y,z;
    }
```

```
	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main(string[] args)

		{
			//
			// TODO: Add code to start application here
		CA obj=new CA();
		Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj.x,obj.y,obj.z);
			//
		}
	}
}
```

Добавим в определение класса CA конструктор по умолчанию:

```
	class CA
	{
		public int x,y,z;
		public CA()
		{
			x=3;
			y=2;
			z=x+y;
		}
	}
```

Конструктор с аргументами

Большинство конструкторов в C# принимают аргументы, с помощью которых разные объекты данного класса могут быть по разному инициализированы.

Пример

```
using System;

namespace ConsoleApplication15
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>

	class CA
	{
		public int x,y,z;
		public CA()
		{
			x=3;
```

```
                y=2;
                z=x+y;
            }
            public CA(int a,int b)
            {
                x=a;
                y=b;
                z=a+b;
            }
            public CA(int a,int b,int c)
            {
                x=a;
                y=b;
                z=c;
            }

        }

        class Class1
        {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                //
                // TODO: Add code to start application here
            CA obj=new CA();
            Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj.x,obj.y,obj.z);
            CA obj1=new CA(5,7);
            Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj1.x,obj1.y,obj1.z);
            CA obj2=new CA(5,7,25);
            Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj2.x,obj2.y,obj2.z);

                //
            }
        }
}
```

Поля только для чтения

Классы могут содержать поля только для чтения. Поле только для чтения – это константное поле, значение которого изменить нельзя. В отличие от обычных констант, значение которых задается непосредственно в тексте программы, начальное значение поля только для чтения может быть вычислено в процессе выполнения приложения. Начальное значение поля только для чтения может быть установлено только внутри конструктора. Для объявления поля для чтения используется ключевое слово readonly.

Пример:

```
using System;

namespace ConsoleApplication15
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>

	class CA
	{
		public int x,y,z;
		public readonly int rd;
		public CA()
		{
			x=3;
			y=2;
			z=x+y;
			rd=x+y+z;

		}
		public CA(int a,int b)
		{
			x=a;
			y=b;
			z=a+b;
			rd=x+y+z;

		}
		public CA(int a,int b,int c)
		{
			x=a;
			y=b;
			z=c;
			rd=x+y+z;
		}

	}

	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
		static void Main(string[] args)
		{
			//
			// TODO: Add code to start application here
		CA obj=new CA();
		Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj.x,obj.y,obj.z);
		CA obj1=new CA(5,7);
		Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj1.x,obj1.y,obj1.z);
		CA obj2=new CA(5,7,25);
```

```
            Console.WriteLine("x={0,2} y={1,2}
z={2,2}",obj2.x,obj2.y,obj2.z);
            Console.WriteLine("поля для чтения{0,2}  {1,2} {2,2}",
obj.rd, obj1.rd, obj2.rd);
                    //
            }
      }
}
```

Интерфейсы

Интерфейс (interface) представляет собой именованный набор абстрактных членов. Абстрактные методы являются чистым протоколом, поскольку не имеют никакой стандартной реализации. Конкретные члены, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. Каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, поддерживать множество поведений.

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.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс — множество методов.

Интерфейсы объявляются с помощью ключевого слова interface. Ниже приведена упрощенная форма объявления интерфейса:

```
interface имя{
    возвращаемый_тип имя_метода_1 (список_параметров);
    возвращаемый_тип имя_метода_2 (список_параметров);
    // ...
    возвращаемый_тип имя_метода_N (список_параметров);
}
```

где имя — это конкретное имя интерфейса. В объявлении методов интерфейса используются только их возвращаемый_тип и сигнатура. Они, по существу, являются абстрактными методами.

Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события. Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как static.

Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе:

```
class имя_класса : имя_интерфейса {
    // тело класса
}
```

где имя_интерфейса — это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

```
// Создаем нтерфейс, описывающий абстрактные методы
// арифметических операций
public interface IArOperation
{
    // Определяем набор абстрактных методов
    int Sum();
    int Otr();
    int Prz();
    int Del();
}

// Данный класс реализует интерфейс IArOperation
class A : IArOperation
{
    int My_x, My_y;

    public int x
    {
        set { My_x = value; }
        get { return My_x; }
    }

    public int y
    {
        set { My_y = value; }
        get { return My_y; }
    }

    public A() { }
    public A(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Реализуем методы интерфейса
    public virtual int Sum()
    {
        return x + y;
    }

    public int Otr()
    {
        return x - y;
    }
```

```
    public int Prz()
    {
        return x * y;
    }

    public int Del()
    {
        return x / y;
    }

    // В данном классе так же можно реализовать собственные методы
    public virtual void rewrite()
    {
        Console.WriteLine("Переменная x: {0}\nПеременная y: {1}", x, y);
    }
}

// Данный класс унаследован от класса A, ри этом в нем не нужно
// заново реализовывать интерфейс,  можно переопределить
// некоторые его методы
class Aa : A
{
    public int z;

    public Aa(int z, int x, int y)
        : base(x, y)
    {
        this.z = z;
    }

    // Переопределим метод Sum
    public override int Sum()
    {
        return base.x + base.y + z;
    }

    public override void rewrite()
    {
        base.rewrite();
        Console.WriteLine("Переменная z: " + z);
    }
}


class Program
{
    static void Main()
    {
        A obj1 = new A(x: 10, y: 12);
        Console.WriteLine("obj1: ");
        obj1.rewrite();
        Console.WriteLine("{0} + {1} = {2}", obj1.x, obj1.y, obj1.Sum());
        Console.WriteLine("{0} * {1} = {2}", obj1.x, obj1.y, obj1.Prz());
    }
}
```

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов.

Таким образом, интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, в иерархии интерфейсов, когда какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своего родителя (или родителей). В отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов.