Лабораторная работа №4

Интерфейсы. Универсальные шаблоны в C#.

1) Интерфейсы

Если требуется наделить какой-либо класс поведением, свойствами различных сущностей, множественное наследование в языке C# можно заменить применением интерфейсов. Интерфейсы схожи с абстрактными классами, у которых все методы абстрактны, однако классы могут реализовывать несколько интерфейсов, тогда как наследовать можно только от одного абстрактного класса. Интерфейсы определяются с помощью ключевого слова interface, как показано в следующем примере.

```
interface IToIntConvertable
{
    int ToInt();
}
```

Интерфейсы описывают группу связанных функциональных возможностей, которые могут принадлежать к любому классу или структуре. Интерфейсы могут содержать методы, свойства, события, индексаторы или любое сочетание этих перечисленных типов членов. Интерфейсы не могут содержать поля. Члены интерфейсов автоматически являются открытыми.

Когда говорят, что класс наследует интерфейс, это означает, что класс предоставляет реализацию для всех членов, определяемых интерфейсом. Сам интерфейс не предоставляет функциональных возможностей, которые класс может наследовать таким же образом, каким могут наследоваться функциональные возможности базового класса. Однако если базовый класс реализует интерфейс, производный класс наследует эту реализацию.

Классы и структуры могут быть унаследованы от интерфейса таким же образом, как классы могут быть унаследованы от базового класса или структуры, но есть два исключения:

1) Класс или структура может наследовать несколько интерфейсов.

2) Когда класс или структура наследует интерфейс, наследуются только имена и подписи методов, поскольку сам интерфейс не содержит реализаций.

Пример реализации интерфейса:

```
class Human : IToIntConvertable
{
    private String mName;
    private String mSurname;
    public int Age { get; set; }     //Свойство Возраст
    public Human(String name, String surname)
    {
        mName = name;
        mSurname = surname;
    }

    public override String ToString()
    {
        return mSurname + " " + mName;
    }

    public int ToInt()
    {
        return Age;
    }
}
```

В этой реализации метод ToInt() возвращает возраст человека. Приведем еще один пример использования интерфейсов:

```
interface IDrawable
{
    void draw(Graphics g);
}

interface IMoveable
{
    void move(int dx, int dy);

    int x { get; }
    int y { get; }
}

abstract class Shape: IMoveable
{
    public int width { get; protected set; }
    public int height { get; protected set; }

    public void move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }

    public int x { get; protected set; }
    public int y { get; protected set; }
}

class Circle : Shape, IDrawable
{
    public void draw(Graphics g)
    {
        // вывод на экран
        g.DrawEllipse(Pens.Black, x, y, width, height);
    }
}
```

Класс Circle наследуется от абстрактного базового класса фигура (Shape), а также реализует интерфейс IDrawable, указывающий на то, что объект может

быть нарисован на экране. Класс Shape, в свою очередь, реализует интерфейс IMoveable, указывающий на то, что все фигуры можно перемещать. Поскольку перемещаемый объект должен иметь координаты, интерфейс IMoveable предполагает наличие двух свойств x, y, доступных для чтения. При реализации этого интерфейса класс, ограничивает доступ к этим свойствам только для наследников класса (атрибут protected).

2) Универсальные шаблоны

Универсальные шаблоны были добавлены в язык C# версии 2.0. Универсальные шаблоны в платформе .NET Framework представляют концепцию параметров типов, которые позволяют разрабатывать классы и методы, не придерживающиеся спецификации одного или нескольких типов до тех пор, пока класс или метод не будет объявлен клиентским кодом и пока не будет создан его экземпляр, как показано в примере:

```
// Объявление универсального шаблона GenericList.
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Объявление списка типов int
        GenericList<int> list1 = new GenericList<int>();

        // Объявление списка типов string
        GenericList<string> list2 = new GenericList<string>();

        // Объявление списка типов ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
    }
}
```

В определении универсального типа или метода параметры типа представляют собой заполнитель для определенного типа, задаваемого клиентом при создании переменной универсального типа. Универсальный класс, такой как GenericList<T>, нельзя использовать "как есть", поскольку он является не типом а, скорее, чертежом типа. Для работы с GenericList<T> в клиентском коде необходимо объявить и создать конструируемый тип, указав в угловых скобках аргумент типа. Аргумент-тип для этого конкретного класса

может быть любым типом, распознаваемым компилятором.

3) Создание коллекции. Метод GetEnumerator(), ключевое слово yield

В подавляющем большинстве случаев универсальные шаблоны используются для создания коллекций элементов произвольного типа данных. Рассмотрим создание коллекции на примере.

```
// Объявление универсального шаблона список, тип данных обозначен как T
public class GenericList<T>
{
    // Скрытый класс Node (элемент) также работает с типом данных T
    private class Node
    {
        // Контруктор класса Node.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next; // Скрытая ссылка на следующий элемент
        public Node Next // Свойство «следующий элемент»
        {
            get { return next; }
            set { next = value; }
        }

        // Переменная типа T, хранит данные
        private T data;
        // Свойство для получения и установки данных типа T в элементе
        public T Data
        {
            get { return data; }
            set { data = value;}
        }
    }

    private Node head; //Хранит ссылку на последний элемент в списке
    // Конструктор списка
    public GenericList()
    {
        head = null;
    }

    // Метод добавления элемента T в начало списка
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    //Метод получения перечислителя. Нужен для foreach
    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
```

```
            current = current.Next;
        }
    }
}
```

Следующий пример кода показывает, как клиентский код использует класс GenericList<T> для создания списка целых чисел. Благодаря простому изменению аргумента-типа, следующий код можно легко преобразовать для создания списка строк или любого другого пользовательского типа.

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Рассмотрим все по порядку. В универсальном шаблоне GenericsList объявлен закрытый класс Node, который хранит данные типа T (поле data), и ссылку на следующий элемент типа Node. Класс имеет свойства для получения и установки данных и ссылки на следующий элемент. Класс имеет единственный конструктор, который принимает в качестве аргумента входные данные элемента и сбрасывает ссылку на следующий элемент.

Далее в шаблоне GenericsList объявлена ссылка на последний элемент списка (head, тип поля Node). Метод AddHead(T t) добавляет элемент t в начало списка. Для этого создается экземпляр класса Node, который становится новым началом списка, а ссылка на текущий элемент-заголовок сохраняется в его поле next.

Метод GetEnumerator() возвращает перечислитель (интерфейс) IEnumerator<T> для списка, и является необходимым для обхода коллекции с помощью цикла foreach. Реализацию методов интерфейса IEnumerator<T> неявно осуществляет ключевое слово *yield*. Работает этот код следующим образом. Текущий элемент current устанавливается в начало списка. Если

ссылка на текущий элемент не нулевая, то из этого элемента, с помощью конструкции *yield return* извлекаются данные, а текущим элементом становится следующий. Процесс повторяется, пока не будет достигнут конец списка и все данные возвращаются в виде коллекции. Для работы с этой коллекцией используется ключевые слова foreach и in:

```
foreach (Class instance in collection)
{

}
```

Здесь в теле цикла foreach создается экземпляр instance класса Class, который по очереди указывает на все элементы коллекции collection.

4) Ограничение параметров типа. Ключевое словоwhere

При определении универсального типа можно ограничить виды типов, которые могут использоваться клиентским кодом в качестве аргументов типа при инициализации соответствующего класса. При попытке клиентского кода создать экземпляр класса с помощью типа, который не допускается ограничением, в результате возникает ошибка компиляции. Это называется ограничениями. Ограничения определяются с помощью контекстно-зависимого ключевого слова *where*:

1) where T : new(). Аргумент типа должен иметь открытый конструктор без параметров. При использовании с другими ограничениями ограничение new() должно устанавливаться последним.

2) where T : base_class_name. Аргумент типа должен являться или быть производным от указанного базового класса.

3) where T : interface_name. Аргумент типа должен являться или реализовывать указанный интерфейс. Можно установить несколько ограничений интерфейса. Ограничивающий интерфейс также может быть универсальным.

Примеры использования:

```
public class GenericList<T> where T : IComparable<T>
{
}

public class GenericList<T> where T : IEquatable<T>
{
```

```
}
```

В первом случае элементами списка могут быть только типы, реализующие интерфейс IComparable<T>. Этот интерфейс определяет обобщенный метод сравнения, тип значения или класс которого используется для создания метода сравнения упорядоченных экземпляров. Интерфейс содержит единственный метод public int CompareTo(T other), который при реализации должен возвращать -1 если значение этого объекта меньше значения other, 0 если значения равны, +1 если значение этого объекта больше значения other. Отметим, что базовые типы double, int, string реализуют указанный интерфейс. Пример реализации интерфейса IComparable<T>:

```
public class Temperature : IComparable<Temperature>
{
    protected double m_value = 0.0;

    public int CompareTo(Temperature other)
    {
        if (other == null) return 1;
        return m_value.CompareTo(other.m_value);
    }
}
```

Интерфейс IEquatable<T> несколько проще интерфейса IComparable<T> и определяет обобщенный метод сравнения, тип значения или класс которого используется для определения равенства экземпляров. Интерфейс имеет единственный метод bool Equals(T other) который при реализации должен возвращать true, если текущий объект равен T, и false в противном случае.

Отметим, что язык C# имеет несколько встроенных универсальных шаблонов для реализации следующих коллекций:

1) List<T> – Представляет строго типизированный список объектов, доступных по индексу. Поддерживает методы для поиска по списку, выполнения сортировки и других операций со списками.

2) Dictionary<TKey, TValue> – Представляет коллекцию ключей и значений.

3) Queue<T> - Представляет коллекцию объектов, основанную на принципе "первым поступил – первым обслужен".

4) Stack<T> – Представляет коллекцию переменного размера экземпляров того же произвольного типа, имеющую тип "последним пришел - первым вышел" (LIFO).

5) Задания к лабораторной работе №4

1) Реализовать класс GenericsFIFO представляющий коллекцию переменного размера для произвольного типа данных (стек FIFO). Реализовать методы добавления элемента в конец списка (Push), и изъятия элемента из начала списка (Pop), получения произвольного по счету элемента без изъятия его из списка (Get). Реализовать свойство, возвращающее общее число элементов в списке.

2) Реализовать класс GenericLIFO представляющий коллекцию переменного размера для произвольного типа данных (стек LIFO). Реализовать методы добавления элемента в начало списка (Push), и изъятия элемента из начала списка (Pop), получения произвольного по счету элемента без изъятия его из списка (Get). Реализовать свойство, возвращающее общее число элементов в списке.

3) Реализовать класс GenericsRing представляющий коллекцию постоянного размера для произвольного типа данных (кольцевой список). Реализовать методы добавления элемента в список (Push), и получения элемента из текущей позиции списка (Head), получения произвольного по счету элемента (Get). Реализовать свойство, возвращающее общее число элементов в списке.

4) Реализовать класс GenericsAscSortedList представляющий сортированный по возрастанию двусвязный список для типов данных, реализующих интерфейс IComparable<T>. Реализовать метод Add добавляющий элемент в нужную позицию в списке, а также методы получения максимального (Max), и минимального (Min) и произвольного (Get) элементов. Реализовать свойство, возвращающее общее число элементов в списке.

5) Реализовать класс GenericsDescSortedList представляющий сортированный по убыванию двусвязный список для типов данных, реализующих интерфейс IComparable<T>. Реализовать метод Add добавляющий элемент в нужную позицию в списке, а также методы получения максимального (Max), и минимального (Min) и произвольного (Get) элементов. Реализовать свойство, возвращающее общее число элементов в списке.

6) Реализовать класс GenericsSet представляющий двусвязный список уникальных элементов, реализующих интерфейс IEquatable<T>. Реализовать метод Add добавляющий уникальный элемент, а также методы получения n-го (Get) элемента. Реализовать свойство, возвращающее общее число элементов в списке.

7) Реализовать класс, представляющий собой коллекцию типа FEFO переменного размера для произвольного типа данных, реализующих ваш собственный интерфейс IExpired (интерфейс требует, чтобы реализующие его классы имели срок годности (типа DateTime) и реализовывали метод int GetExpireDays(), возвращающий количество оставшихся на сегодняшний день дней). В классе коллекции должен быть метод Add(T t) добавляющий элемент в список (если у элемента уже истёк срок годности, то он в список не добавляется). Так же должны быть методы для изъятия элемента из начала (PopBad) и конца списка (PopFresh). Так же, реализовать свойство, возвращающее общее число элементов в списке.

6) Контрольные вопросы

1) Что такое интерфейс и для чего он предназначен?

2) Что, как правило, требуется от программиста при наследовании интерфейса?

3) Опишите интерфейс IEquatable.

4) Опишите интерфейс IComparable. В каких случаях он применяется.

5) Какие основные разновидности коллекций существуют в C#?