УО «МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ А.А. КУЛЕШОВА»
СОЦИАЛЬНО-ГУМАНИТАРНЫЙ КОЛЛЕДЖ

Дисциплина
«Конструирование программ и языки программирования»

Разработка программ с использованием интерфейсов

(4 часа)

Методические рекомендации к лабораторной работе №11

Могилев 2018

Понятие «интерфейс». Методические указания по лабораторной работе №11 «Конструирование программ и языки программирования». Для учащихся очной формы обучения специальности 1–40 01 01 «Программное обеспечение информационных технологий.

Содержание

1 Цель работы

Цель:

изучить описание и работу интерфейсов.

2 Ход работы

1. Изучение теоретического материала.
2. Выполнение практических индивидуальных заданий по вариантам (вариант уточняйте у преподавателя).
3. Оформление отчета.
   3.1. Отчет оформляется индивидуально каждым студентом. Отчет должен содержать задание, алгоритм и листинг программы.
   3.2. Отчет по лабораторной работе выполняется на листах формата А4. В состав отчета входят:
      1) титульный лист;
      2) цель работы;
      3) текст индивидуального задания;
      4) выполнение индивидуального задания.
4. Контрольные вопросы.

3 Краткие теоретические сведения

3.1 Интерфейсы

Используя механизм наследования, мы можем дополнять и переопределять общий функционал базовых классах в классах-наследниках. Однако напрямую мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке C# подобную проблему позволяют решить интерфейсы. Они играют важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы.

Для определения интерфейса используется ключевое слово `interface`. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, `IComparable`, `IEnumerable` (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Интерфейсы так же, как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Определим следующий интерфейс `IAccount`, который будет содержать методы и свойства для работы со счетом клиента. Для добавления интефрейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать `Add-> New Item` и в диалоговом окне добавления нового компонента выбрать `Interface`:

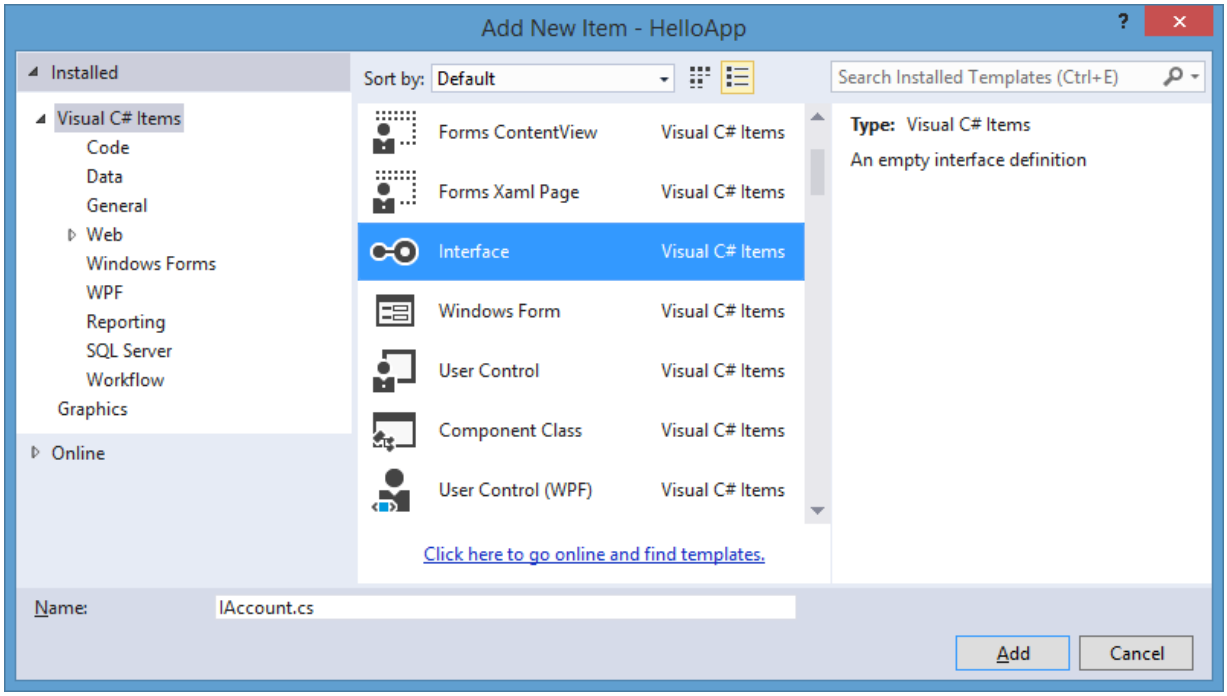

Рис. 3.1 Интерфейсы в C#

Изменим пустой код интерфейса `IAccount` на следующий:

```
interface IAccount
{
    // Текущая сумма на счету
    int CurrentSum { get; }
```

```
    // Положить деньги на счет
    void Put(int sum);
    // Взять со счета
    void Withdraw(int sum);
    // Процент начислений
    int Percentage { get; }
}
```

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет два свойства для текущей суммы денег на счете и ставки процента по вкладам и два метода для добавления денег на счет и изъятия денег.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ `public`, так как цель интерфейса – определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Применение интерфейса аналогично наследованию класса:

```
class Client : IAccount
{
// реализация методов и свойств интерфейса
}
```

Теперь же реализуем интерфейс в классе Client, так как клиент у нас обладает счетом:

```
class Client : IAccount
{
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента

    public string Name { get; set; }
    public Client(string name, int sum, int percentage)
    {
        Name = name;
        _sum = sum;
        _percentage = percentage;
    }
    public int CurrentSum
    {
        get { return _sum; }
    }
    public void Put(int sum)
    {
        _sum += sum;
    }
```

```
        public void Withdraw(int sum)
        {
            if (sum <= _sum)
            {
                _sum -= sum;
            }
        }
        public int Percentage
        {
            get { return _percentage; }
        }
        public void Display()
        {
            Console.WriteLine("Клиент " + Name + " имеет счет на сумму
" + _sum);
        }
    }
```

Как и в случае с абстрактными методами абстрактного классом класс `Client` реализует все методы интерфейса. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор `public`. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, `protected`, то интерфейс не подходит для определения подобного метода.

Применение класса в программе:

```
Client client = new Client("Tom", 200, 10);
client.Put(30);
Console.WriteLine(client.CurrentSum); //230
client.Withdraw(100);
Console.WriteLine(client.CurrentSum); //130

Интерфейсы, как и классы, могут наследоваться:

interface IDepositAccount : IAccount
{
    void GetIncome(); // начисление процентов
}
```

При применении этого интерфейса класс `Client` должен будет реализовать как методы и свойства интерфейса `IDepositAccount`, так и методы и свойства базового интерфейса `IAccount`.

3.2 Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс `Client` реализует интерфейс `IAccount`, то переменная типа `IAccount` мо-

жет хранить ссылку на объект типа `Client`:

```
IAccount client1 = new Client("Том", 200, 10);
client1.Put(200);
Console.WriteLine(client1.CurrentSum); // 400
// Интерфейс не имеет метода Display, необходимо явное приведение
((Client)client1).Display();
```

И если мы хотим обратиться к методам класса `Client`, которые не определены в интерфейсе `IAccount`, а определены в самом классе `Client` или в его базовом классе, то нам надо явным образом выполнить преобразование типов:
`((Client)client1).Display();`

3.3 Обобщенные интерфейсы

Как и классы, интерфейсы могут быть обобщенными:

```
interface IAccount<T>
{
    void SetSum(T _sum);
    void Display();
}
class Client<T> : IAccount<T>
{
    T sum=default(T);
    public void SetSum(T _sum)
    {
        this.sum = _sum;
    }
    public void Display()
    {
        Console.WriteLine(sum);
    }
}
```

Интерфейс `IAccount` типизирован параметром `T`, который при реализации интерфейса используется в классе `Client`. В частности переменная суммы определена как `T`, что позволяет нам использовать для суммы различные числовые типы.

Определим две реализации: одна в качестве параметра будет использовать тип `int`, а другая - тип `double`:

```
IAccount<int> intClient = new Client<int>();
intClient.SetSum(300);
intClient.Display();
IAccount<double> doubleClient = new Client<double>();
doubleClient.SetSum(500.09);
doubleClient.Display();
```

3.4 Явное применение интерфейсов

Может сложиться ситуация, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров. Например:

```
class Person : ISchool, IUniversity
{
    public void Study()
    {
        Console.WriteLine("Учеба в школе или в университете");
    }
}
interface ISchool
{
    void Study();
}
interface IUniversity
{
    void Study();
}
```

Класс `Person` определяет один метод `Study()`, создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект `Person` как объект типа `ISchool` или `IUniversity`, результат метода будет один и тот же.

Однако нередко бывает необходимо разграничить реализуемые интерфейсы. В этом случае надо явным образом применить интерфейс:

```
class Person : ISchool, IUniversity
{
    void ISchool.Study()
    {
        Console.WriteLine("Учеба в школе");
    }
    void  IUniversity.Study()
    {
        Console.WriteLine("Учеба в университете");
    }
}
```

При явной реализации указывается название метода вместе с названием интерфейса, при этом мы не можем использовать модификатор `public`, то есть методы являются закрытыми. В этом случае при использовании метода `Study` в программе нам надо объект `Person` привести к типу соответствующего интерфейса:

```
static void Main(string[] args)
{
    Person p = new Person();
    ((ISchool)p).Study();
    ((IUniversity)p).Study();
    Console.Read();
}
```

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности:

```
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; }
}
```

В данном случае объекты `p1` и `p2` будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной `p2` затронут также и переменную `p1`.

Чтобы переменная `p2` указывала на новый объект, но со значениями из `p1`, мы можем применить клонирование с помощью реализации интерфейса `ICloneable`:

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

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

```
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 };
    }
}
```

Использование:

```
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();
    }
}
```

Теперь все нормально копируется, изменения в свойствах `p2` не сказываются на свойствах в `p1`.

Для сокращения кода копирования мы можем использовать специальный метод `MemberwiseClone()`, который возвращает копию объекта:

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

Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:

```
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; }
    }
```

В этом случае при копировании новая копия будет указывать на тот же объект `Company`:

```
    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
```

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:

```
    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; }
    }
```

3.5 Интерфейс IComparable

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется `Sort()` можно сразу отсортировать по возрастанию весь набор данных. Например:

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

Однако метод `Sort` по умолчанию работает только для наборов примитивных типов, как `int` или `string`. Для сортировки наборов сложных объектов применяется интерфейс `IComparable`. Он имеет всего один метод:

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

Метод `CompareTo` предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

- Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
- Равен нулю. Значит, оба объекта равны
- Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра.

Например, имеется класс `Person`:

```
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");
    }
}
```

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

```
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 IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

```
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);
    }
}
```

3.6 Применение компаратора

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

```
interface IComparer
{
    int Compare(object o1, object o2);
}
```

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

```
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;
    }
}
```

В данном случае используется обобщенная версия интерфейса `IComparer`, чтобы не делать излишних преобразований типов. Применение компаратора:

```
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);
}
```

Объект компаратора указывается в качестве второго параметра метода `Array.Sort()`. При этом не важно, реализует ли класс `Person` интерфейс `IComparable` или нет. Правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты `Person`, у которых имена меньше, а в конце - у которых имена длиннее:

```
Tom - 23
Bill - 34
Alice - 21
```

4 Задания

Во всех классах из ЛР №10 (в соответствии с вариантом) реализовать интерфейсов IComparable и `IComparer` и перегрузить операции отношения (см. лекции) для реализации значимой семантики сравнения объектов по какому-либо полю на усмотрение учащегося.

5 Контрольные вопросы

1. Что понимается под термином «интерфейс»?
2. Чем отличается синтаксис интерфейса от синтаксиса абстрактного класса?
3. Какое ключевое слово языка C# используется для описания интерфейса?
4. Поддерживают ли реализацию методы интерфейса?
5. Какие объекты языка C# могут быть членами интерфейсов?
6. Каким количеством классов может быть реализован интерфейс?
7. Может ли класс реализовывать множественные интерфейсы?
8. Необходима ли реализация методов интерфейса в классе, включающем этот интерфейс?
9. Какой модификатор доступа соответствует интерфейсу?
10. Допустимо ли явное указание модификатора доступа для интерфейса?
11. Приведите синтаксис интерфейса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
12. Возможно ли создание ссылочной переменной интерфейсного типа?
13. Возможно ли наследование интерфейсов?
14. Насколько синтаксис наследования интерфейсов отличается от синтаксиса наследования классов?
15. Необходимо ли обеспечение реализации в иерархии наследуемых интерфейсов?