Лабораторная работа № 5

**Интерфейсы.**

**Цель лабораторной работы**: научиться создавать и использовать интерфейсы.

**Теоретическая часть**

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 С++, 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 также, 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:

|  |  |
| --- | --- |
| **interface** IAccount  **{**  // Текущая сумма на счету  int CurrentSum **{** get**;** **}**  // Положить деньги на счет  void Put**(**int sum**);**  // Взять со счета  void Withdraw**(**int sum**);**  **}** |  |

У **интерфейса методы и свойства не имеют реализации**, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет свойство для текущей суммы денег на счете 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.

Подобным образом добавим в проект еще один интерфейс, который назовем **IClient**:

|  |  |
| --- | --- |
| **interface** IClient  **{**  string Name **{** get**;** set**;** **}**  **}** |  |

Применение интерфейса аналогично наследованию класса:

|  |  |
| --- | --- |
| class Client **:** IAccount  **{**  // реализация методов и свойств интерфейса  **}** |  |

Через запятую можно применить несколько интерфейсов:

|  |  |
| --- | --- |
| class Client **:** IAccount**,** IClient  **{**  // реализация методов и свойств интерфейса  **}** |  |

Теперь же реализуем интерфейсы в классе Client:

|  |  |
| --- | --- |
| class Client **:** IAccount**,** IClient  **{**  int \_sum**;** // Переменная для хранения суммы    **public** string Name **{** get**;** set**;** **}**  **public** Client**(**string name**,** int sum**)**  **{**  Name **=** name**;**  \_sum **=** sum**;**  **}**    **public** int CurrentSum  **{**  get **{** **return** \_sum**;** **}**  **}**    **public** void Put**(**int sum**)**  **{**  \_sum **+=** sum**;**  **}**    **public** void Withdraw**(**int sum**)**  **{**  **if** **(**sum **<=** \_sum**)**  **{**  \_sum **-=** sum**;**  **}**  **}**  **}** |  |

Как и в случае с абстрактными методами абстрактного классом класс Client реализует все методы и свойства 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**);**  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.

**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**();**

**}**

**Интерфейсы как ограничения обобщений**

Интерфейсы могут выступать в качестве ограничений обобщений. При этом если в качестве ограничения можно указаь только один класс, то интерфейсов можно указать несколько.

Допустим, у нас есть следующие интерфейсы и класс, который их реализует:

**interface** IAccount

**{**

int CurrentSum **{** get**;** **}** // Текущая сумма на счету

void Put**(**int sum**);** // Положить деньги на счет

void Withdraw**(**int sum**);** // Взять со счета

**}**

**interface** IClient

**{**

string Name **{** get**;** set**;** **}**

**}**

class Client **:** IAccount**,** IClient

**{**

int \_sum**;** // Переменная для хранения суммы

**public** Client**(**string name**,** int sum**)**

**{**

Name **=** name**;**

\_sum **=** sum**;**

**}**

**public** string Name **{** get**;** set**;** **}**

**public** int CurrentSum

**{**

get **{** **return** \_sum**;** **}**

**}**

**public** void Put**(**int sum**)**

**{**

\_sum **+=** sum**;**

**}**

**public** void Withdraw**(**int sum**)**

**{**

**if** **(**sum **<=** \_sum**)**

**{**

\_sum **-=** sum**;**

**}**

**}**

**}**

Используем выше перечисленные интерфейсы в качестве ограничений обобщенного класса:

class Transaction**<**T**>** where T**:** IAccount**,** IClient

**{**

**public** void Operate**(**T acc1**,** T acc2**,** int sum**)**

**{**

**if(**acc1**.**CurrentSum **>=** sum**)**

**{**

acc1**.**Withdraw**(**sum**);**

acc2**.**Put**(**sum**);**

Console**.**WriteLine**(**$"{acc1.Name} : {acc1.CurrentSum}\n{acc2.Name} : {acc2.CurrentSum}"**);**

**}**

**}**

**}**

В данном случае параметр T представляет тип, который который реализует сразу два интерфейса IAccount и IClient. Например, выше определен класс Client, который реализует оба интерфейса, поэтому мы можем данным типом типизировать объекты Transaction:

Client account1 **=** **new** Client**(**"Tom"**,** 200**);**

Client account2 **=** **new** Client**(**"Bob"**,** 300**);**

Transaction**<**Client**>** transaction **=** **new** Transaction**<**Client**>();**

transaction**.**Operate**(**account1**,** account2**,** 150**);**

Также параметр T может представлять интерфейс, который наследуется от обоих интерфейсов:

**interface** IClientAccount **:** IAccount**,** IClient

**{**

**}**

class ClientAccount **:** IClientAccount

**{**

int \_sum**;**

**public** ClientAccount**(**string name**,** int sum**)**

**{**

\_sum **=** sum**;** Name **=** name**;**

**}**

**public** int CurrentSum **{** get **{** **return** \_sum**;** **}** **}**

**public** string Name **{** get**;** set**;** **}**

**public** void Put**(**int sum**)**

**{**

\_sum **+=** sum**;**

**}**

**public** void Withdraw**(**int sum**)**

**{**

**if** **(**\_sum **>=** sum**)** \_sum **-=** sum**;**

**}**

**}**

В этом случае объекты Transaction мы можем типизировать типом IClientAccount:

IClientAccount account3 **=** **new** ClientAccount**(**"Alice"**,** 400**);**

IClientAccount account4 **=** **new** ClientAccount**(**"Kate"**,** 500**);**

Transaction**<**IClientAccount**>** operation **=** **new** Transaction**<**IClientAccount**>();**

operation**.**Operate**(**account3**,** account4**,** 200**);**

**Обобщенные интерфейсы**

Как и классы, интерфейсы могут быть обобщенными:

**interface** IUser**<**T**>**

**{**

T Id **{** get**;** **}**

**}**

class User**<**T**>** **:** IUser**<**T**>**

**{**

T \_id**;**

**public** User**(**T id**)**

**{**

\_id **=** id**;**

**}**

**public** T Id **{** get **{** **return** \_id**;** **}** **}**

**}**

Интерфейс IUser типизирован параметром T, который при реализации интерфейса используется в классе User. В частности, переменная \_id определена как T, что позволяет нам использовать для id различные типы.

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

IUser**<**int**>** user1 **=** **new** User**<**int**>(**6789**);**

Console**.**WriteLine**(**user1**.**Id**);** // 6789

IUser**<**string**>** user2 **=** **new** User**<**string**>(**"12345"**);**

Console**.**WriteLine**(**user2**.**Id**);** // 12345

Также при реализации интерфейса мы можем явным образом указать, какой тип будет использоваться для параметра T:

class IntUser **:** IUser**<**int**>**

**{**

int \_id**;**

**public** IntUser**(**int id**)**

**{**

\_id **=** id**;**

**}**

**public** int Id **{** get **{** **return** \_id**;** **}** **}**

**}**

**Клонирование объектов. Интерфейс ICloneable**

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**;** **}**

**}**

**Сортировка объектов. 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**);**

**}**

**}**

**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

**Практическая часть**

**Задание 1.**

Создать иерархию классов:

Человек

Студент

Класс «человек» должен содержать следующие элементы: поле-фамилия, поле-год рождения, поле статус (**студент, преподаватель, бизнесмен и т.д.**), конструктор с параметрами**,** конструктор без параметров, метод вывода информации об объекте, **виртуальный** методом **Info** возвращающим возраст. Использовать метод **Info** в методе вывода информации. Реализовать в классе «Человек» интерфейс **IComparable** длясравнения людей по возрасту.

Класс студентов должен содержать дополнительные поля-оценки по математике, физике и истории, конструктор с параметрами, метод для получения среднего балла за сессию. Переопределить метод **Info** в классе «Студент» так, чтобы он возвращал максимальную оценку.

Дополнительно создать класс , реализующий интерфейс **IComparer.** Использовать объект класса для сортировки списка людей по алфавиту.

В методе Main:

* формировать массив объектов класса человек (с клавиатуры не вводить, генератор случайных значений не использовать).
* Сортировать по возрасту.
* Сортировать по фамилии.
* Выводить информацию

**Задание 2.**

Создать классы квадратов и окружностей, реализующие общий интерфейс «Геометрические фигуры»:

интерфейс Геометрическая фигура

Геометрическая фигура

Квадрат

Круг

Интерфейс должен определять следующие элементы: свойство, возвращающее площадь фигуры, метод вывода информации.

Класс квадратов должен содержать следующие элементы: поле-сторону квадрата, конструктор, реализованные элементы интерфейса, метод вычисления периметра. Класс кругов должен содержать следующие элементы: поля - радиус, цвет фигуры, конструктор, реализованные элементы интерфейса.

Создать класс-контейнер для фигур, содержащий поле-массив фигур (ссылок интерфейсного типа), конструктор и реализующий интерфейс IEnumerable.

В классе Program создать объект класса-контейнера с информацией о двух кругах и двух квадратах. Вывести информацию о фигурах, используя для просмотра содержимого контейнера оператор **foreach**.