Лабораторная работа №2

ПОЛИМОРФИЗМ

Цель работы: изучить механизмы реализации полиморфизма в C# и ознакомиться с основными подходами при использовании интерфейсов.

Краткие теоретические сведения

Полиморфизм является третьим ключевым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса, и определение полиморфного интерфейса в базовом классе – набор членов класса, которые могут быть переопределены в классе-наследнике. Методы, которые надо переопределить, в базовом классе помечаются модификатором **virtual** и называются виртуальными. Они и представляют полиморфный интерфейс. Также частью полиморфного интерфейса могут быть абстрактные члены класса.

При определении класса-наследника и наследовании методов базового класса мы можем выбрать одну из следующих стратегий: обычное наследование всех членов базового класса, переопределение членов базового класса или скрытие членов базового класса в классе-наследнике.

Стратегия обычного наследования довольно проста. Допустим, есть следующая пара классов **Person** и **Employee**:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string lName, string fName)
    {
        FirstName = fName;
        LastName = lName;
    }
    public virtual void Display()
    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
        : base(fName, lName)
```

```
    {
        Company = comp;
    }
}
```

В базовом классе **Person** метод **Display()** определён с модификатором **virtual**, поэтому данный метод может быть переопределён. Но класс **Employee** наследует его как есть:

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person("Bill", "Gates");
        p1.Display(); // вызов метода Display из класса Person
        Person p2 = new Employee("Tom", "Johns", "UnitBank");
        p2.Display(); // вызов метода Display из класса Person
        Employee p3 = new Employee("Sam", "Toms", "CreditBank");
        p3.Display(); // вызов метода Display из класса Person
        Console.Read();
    }
}
```

Консольный вывод:

```
Bill Gates
Tom Johns
Sam Toms
```

Переопределение членов базового класса в классе-наследнике предполагает использование ключевого слова **override**:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
        : base(fName, lName)
    {
        Company = comp;
    }

    public override void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + " работает
в компании " + Company);
    }
}
```

Класс **Person** остаётся тем же, метод **Display** также объявляется виртуальным. В этом случае поведение объекта **Employee** измениться:

```
Person p1 = new Person("Bill", "Gates");
```

```
p1.Display(); // вызов метода Display из класса Person
Person p2 = new Employee("Tom", "Johns", "UnitBank");
p2.Display(); // вызов метода Display из класса Employee
Employee p3 = new Employee("Sam", "Toms", "CreditBank");
p3.Display(); // вызов метода Display из класса Employee
```

Консольный вывод:

```
Bill Gates
Tom Johns работает в компании UnitBank
Sam Toms работает в компании CreditBank
```

При скрытии членов базового класса в классе-наследнике можно просто определить в нём метод с тем же именем, без переопределения, используя слово **override**:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
        : base(fName, lName)
    {
        Company = comp;
    }

    public new void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + " работает
в компании " + Company);
    }
}
```

В этом случае метод **Display()** в **Employee** скрывает этот же метод из класса **Person**. Чтобы явно скрыть метод из базового класса, используется ключевое слово **new**, хотя в принципе оно необязательно, по умолчанию система это делает неявно.

Приведём пример использования ключевого слова **new** в программе:

```
Person p1 = new Person("Bill", "Gates");
p1.Display(); // вызов метода Display из класса Person
Person p2 = new Employee("Tom", "Johns", "UnitBank");
p2.Display(); // вызов метода Display из класса Person
Employee p3 = new Employee("Sam", "Toms", "CreditBank");
p3.Display(); // вызов метода Display из класса Employee
```

Консольный вывод:

```
Bill Gates
Tom Johns
Sam Toms работает в компании CreditBank
```

Следует обратить внимание на различия в вызовах метода **Display** из класса **Person**.

Кроме конструкторов, с помощью ключевого слова **base** можно обратиться к другим членам базового класса. Таким образом, вызов **base.Display()** будет обращением к методу **Display()** в классе **Person**:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
        : base(fName, lName)
    {
        Company = comp;
    }
    public override void Display()
    {
        base.Display();
        Console.WriteLine("Место работы : " + Company);
    }
}
```

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определённую версию метода. Допустим, что есть следующий класс **State**:

```
class State
{
    public string Name { get; set; } // название
    public int Population { get; set; } // население
    public double Area { get; set; } // площадь
}
```

Необходимо определить метод для нападения на другое государство – метод **Attack**. Первая реализация этого метода будет принимать в качестве параметра объект **State** – это государство, на которое нападают:

```
public void Attack(State enemy)
{ }
```

Предположим, что надо определить версию данного метода, где будет указано не только государство, но и количество войск. Тогда можно будет просто добавить вторую версию данного метода:

```
public void Attack(State enemy)
{
    // здесь код метода
}
```

```
public void Attack(State enemy, int army)
{
    // здесь код метода
}
```

Наряду с методами возможно также перегружать операторы. При этом необходимо указывать модификаторы **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. После модификаторов идёт название возвращаемого типа, и только после него ключевое слово **operator** и лишь затем название оператора и параметры:

```
public static возвращаемый_тип operator оператор(параметры)
```

Рассмотрим на примере. Есть класс **State** – государство. Необходимо объединить несколько государств. В этом случае задачу сможет облегчить перегрузка **operator** +. Кроме того, используем операторы сравнения «>» и «<», с помощью которых будем сравнивать два государства:

```
class State
{
    public string Name { get; set; } // название
    public int Population { get; set; } // население
    public double Area { get; set; } // площадь
    public static State operator +(State s1, State s2)
    {
        string name = s1.Name;
        int people = s1.Population + s2.Population;
        double area = s1.Area + s2.Area;
        // возвращаем новое объединённое государство
        return new State { Name = name, Area = area, Population =
people };
    }
    public static bool operator <(State s1, State s2)
    {
        if (s1.Area < s2.Area)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public static bool operator >(State s1, State s2)
    {
        if (s1.Area > s2.Area)
        {
            return true;
        }
```

```
        else
        {
            return false;
        }
    }
}
```

Поскольку все перегруженные операторы бинарные, т. е. проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра. Теперь используем перегруженные операторы в программе:

```
static void Main(string[] args)
{
    State s1 = new State { Name = "State1", Area = 300, Population
= 100 };
    State s2 = new State { Name = "State2", Area = 200, Population
= 70 };
    if (s1 > s2)
    {
        Console.WriteLine("Государство s1 больше государства s2");
    }
    else if (s1 < s2)
    {
        Console.WriteLine("Государство s1 меньше государства s2");
    }
    else
    {
        Console.WriteLine("Государства s1 и s2 равны");
    }
    State s3 = s1 + s2;
    Console.WriteLine("Название государства : {0}",
   s3.Name);
    Console.WriteLine("Площадь государства : {0}",
   s3.Area);
    Console.WriteLine("Население государства : {0}",
   s3.Population);
    Console.ReadLine();
}
```

Следует учитывать, что не все операторы можно перегружать.

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный. Он также может иметь переменные, методы, конструкторы и свойства. Но нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы лишь предоставляют базовый функционал для классов-наследников, а реализуют его производные классы.

При определении абстрактных классов используется ключевое слово **abstract**:

```
abstract class Human
{
    public int Length { get; set; }
    public double Weight { get; set; }
}
```

Кроме обычных, абстрактный класс может иметь абстрактные методы. Подобные методы также определяются с помощью ключевого слова **abstract** и не имеют никакого функционала:

```
public abstract void Display();
```

При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод также объявляется с модификатором **override**. Также следует учесть, что если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определён как абстрактный.

Абстрактные методы так же, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Зачем нужны абстрактные классы? Допустим, программе для банковского сектора необходимо определить три класса: **person**, который описывает человека; **employee** – сотрудника банка; **client** – клиента банка.

Очевидно, что классы **Employee** и **Client** будут производными от класса **Person**. И так как все объекты будут представлять либо сотрудника банка, либо клиента, то напрямую от класса **Person** создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string lName, string fName)
    {
        FirstName = fName;
        LastName = lName;
    }
    public abstract void Display();
}
```

```
class Client : Person
{
    public string Bank { get; set; }
    public Client(string lName, string fName, string comp)
        : base(fName, lName)
    {
        Bank = comp;
    }
    public override void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + " имеет
счёт в банке " + Bank);
    }
}
```

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Определяются они с помощью ключевого слова **var** и инициализатора объектов:

```
var user = new { Name = "Tom", Age = 34 };
Console.WriteLine(user.Name);
```

В данном случае **user** – это объект анонимного типа с двумя определёнными свойствами **Name** и **Age**, которые можно использовать так же, как и у обычных объектов классов. Однако тут есть ограничение – свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие **"<>f__AnonymousType0'2"**.

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

Если в программе используется несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа:

```
var user = new { Name = "Tom", Age = 34 };
var student = new { Name = "Alice", Age = 21 };
var manager = new
{
    Name = "Bob",
    Age = 26,
    Company = "Microsoft"
};
Console.WriteLine(user.GetType().Name); // <> f__AnonymousType0'2
Console.WriteLine(student.GetType().Name); // <>
f__AnonymousType0'2
```

```
Console.WriteLine(manager.GetType().Name);          //          <>
f__AnonymousType1'3
```

Здесь **user** и **student** будут иметь одно и то же определение анонимного типа. Однако подобные объекты нельзя преобразовать к каком-нибудь другому типу, например классу, даже если он имеет подобный набор свойств.

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если необходимо добавить свойство, то это можно будет сделать сразу на месте анонимного объекта. В случае с классом придётся изменять ещё и класс, который может больше нигде не использоваться. Типичная ситуация – получение результата выборки из базы данных: объекты используются только для получения выборки и больше нигде не используются, поэтому классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

Задание на лабораторную работу

Постройте иерархию классов, используя абстрактный класс и интерфейс в соответствии с вариантом задания. Реализуйте пример использования полиморфизма методов.

Вариант	Задания
1	Студент, преподаватель, персона, заведующий кафедрой
2	Служащий, персона, рабочий, инженер
3	Рабочий, кадры, инженер, администрация
4	Деталь, механизм, изделие, узел
5	Организация, страховая компания, нефтегазовая компания, завод
6	Журнал, книга, печатное издание, учебник
7	Тест, экзамен, выпускной экзамен, испытание
8	Место, область, город, мегаполис
9	Игрушка, продукт, товар, молочный продукт
10	Квитанция, накладная, документ, счёт
11	Автомобиль, поезд, транспортное средство, экспресс
12	Двигатель, дизель, двигатели внутреннего сгорания и реактивный
13	Республика, монархия, королевство, государство
14	Млекопитающее, парнокопытное, птица, животное
15	Товар, велосипед, горный велосипед, самокат
16	Лев, дельфин, птица, синица, животное
17	Музыкант, персона, студент, гитарист
18	Печатное издание, газета, книга, периодика
19	Корабль, пароход, парусник, корвет
20	Стихотворение, стиль изложения, рифма, проза
21	Посёлок, область, район, город

Вариант	Задания
22	Грузовик, автомобиль, легковое авто, транспорт
23	Спорт, футбол, хобби, музыка
24	Молоток, инструмент, гитара, звук
25	Окружность, геометрическая фигура, линия, заливка

Порядок выполнения работы

1. Измените, расширьте и опишите иерархию классов, используя:

1) описание и наследование классами как минимум трёх интерфейсов;

2) виртуальный класс в качестве основы полиморфизма.

2. Покажите на примере одного из методов, присутствующих в каждом классе, свойство полиморфизма.

Контрольные вопросы

1. Что понимается под термином «полиморфизм»?
2. В чём состоит основной принцип полиморфизма?
3. В чём состоит значение основного принципа полиморфизма?
4. Какие механизмы используются в языке C# для реализации концепции полиморфизма?
5. Что понимается под термином «виртуальный метод»?
6. Какое ключевое слово языка C# используется для определения виртуального метода?
7. В чём состоит особенность виртуальных методов в производных (дочерних) классах?
8. В какой момент трансляции программы осуществляется выбор версии виртуального метода?
9. Какие условия определяют выбор версии виртуального метода?
10. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе?
11. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в производном (дочернем) классе?
12. Какие модификаторы недопустимы для определения виртуальных методов?
13. Что означает термин «переопределённый метод»?
14. В какой момент трансляции программы осуществляется выбор вызываемого переопределённого метода?
15. Приведите синтаксис виртуального метода в общем виде.