Лабораторная работа №1

КЛАССЫ И ОБЪЕКТЫ, ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ

Цель работы: овладение навыками проектирования простейших классов и создания объектов класса; освоение принципов инкапсуляции и наследования.

Краткие теоретические сведения

Язык программирования высокого уровня C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описание объекта – класс, а объект – экземпляр этого класса. Можно ещё провести следующую аналогию. У всех есть некоторое представление о человеке – руки, ноги, голова, пищеварительная и нервная система, головной мозг и т. д. – некоторый шаблон. Этот шаблон можно назвать классом. Реально же существующий человек, фактически экземпляр данного класса, является объектом класса.

Класс определяется с помощью ключевого слова **class**:

```
class Book { }
```

Вся функциональность класса представлена его членами – полями (полями называются переменные класса), свойствами, методами, событиями. Для примера создадим класс **Book**, в котором будут храниться переменные – название, автор и год издания книги. Кроме того, класс будет содержать метод для вывода информации о книге на консоль:

```
class Book
{
    public string name;
    public string author;
    public int year;
    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана в
{2} году", name, author, year);
    }
}
```

Кроме обычных методов в классах используются и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Отличительной чертой конструктора является то, что его название должно совпадать с названием класса:

```
class Book
{
    public string name;
    public string author;
    public int year;
    // конструктор без параметров
    public Book()
    { }
    // конструктор с параметрами
    public Book(string name, string author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана в
{2} году", name, author, year);
    }
}
```

Одно из назначений конструктора – начальная инициализация членов класса. В данном случае было использовано два конструктора **public Book**: один – пустой, а второй наполняет поля класса начальными значениями, которые передаются через его параметры.

Поскольку имена параметров и имена полей **name**, **author**, **year** в данном случае совпадают, то мы используем ключевое слово **this**. Это ключевое слово представляет ссылку на текущий экземпляр класса. Поэтому в выражении **this.name = name;** первая часть **this.name** означает, что **name** – это поле текущего класса, а не название параметра **name**. Если бы параметры и поля назывались по-разному, то использование слова **this** было бы необязательно.

Теперь используем класс в программе. Создадим новый проект, а затем нажмём правой кнопкой мыши на название проекта в окне **Solution Explorer** и в появившемся меню выберем пункт **Class**.

В появившемся диалоговом окне дадим новому классу имя **Book** и нажмём кнопку **Add** (Добавить). В проект будет добавлен новый файл **Book.cs**, содержащий класс **Book**.

Изменим в этом файле код класса **Book**:

```
class Book
{
    public string name;
    public string author;
```

```
    public int year;
    public Book()
    {
        name = "неизвестно";
        author = "неизвестно";
        year = 0;
    }
    public Book(string name, string author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }
    public void GetInformation()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана в
{2} году", name, author, year);
    }
}
```

Теперь перейдём к коду файла **Program.cs** и изменим метод **Main** класса **Program** следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой",
       1869);
        b1.GetInformation();
        Book b2 = new Book();
        b2.GetInformation();
        Console.ReadLine();
    }
}
```

Если запустить код на выполнение, то на консоль выведется информация о книгах **b1** и **b2**. Следует обратить внимание, что для того чтобы создать новый объект с использованием конструктора, нам надо использовать ключевое слово **new**. Оператор **new** создаёт объект класса и выделяет для него область в памяти.

Для класса **Book** установим последовательно значения для всех трёх его полей:

```
Book b1 = new Book();
b1.name = "Война и мир";
b1.author = "Л. Н. Толстой";
b1.year = 1869;

b1.GetInformation();
```

Но можно также использовать инициализатор объектов:

```
Book b2 = new Book();
b2 = new Book { name = "Отцы и дети", author = "И. С.Тургенев",
year = 1862 };
b2.GetInformation();
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

Все члены класса – поля, методы, свойства – имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, т. е. контекст, в котором можно употреблять данную переменную или метод. При объявлении полей класса **Book** использовался модификатор **public**, который означает, что все объявленные за ним члены класса общедоступны из любого места в коде, а также из других программ и сборок. Также в C# применяются и другие модификаторы доступа:

- **private** означает закрытый класс или член класса и представляет полную противоположность модификатору **public**, доступны только из кода в том же классе или контексте;
- **protected** означает, что член класса доступен из любого места в текущем классе или в производных классах;
- **internal** означает, что класс и члены класса доступны из любого места кода в той же сборке, но недоступны для других программ и сборок, как в случае с модификатором **public**;
- **protected internal** – совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором **private**. Классы, объявленные без модификатора, по умолчанию имеют доступ **internal**.

Рассмотрим это на примере создания класса **State**:

```
public class State
{
    int a; // всё равно, что private int a;
           // поле доступно только из текущего класса
    private int b;
    // доступно из текущего класса и производных классов
    protected int c;
```

```
    internal int d; // доступно в любом месте программы
                    // доступно в любом месте программы
                    // и из классов-наследников
    protected internal int e;
    // доступно в любом месте программы,
    // а также для других программ и сборок
    public int f;
    private void Display_f()
    {
        Console.WriteLine("Переменная f = {0}", f);
    }
    public void Display_a()
    {
        Console.WriteLine("Переменная a = {0}", a);
    }
    internal void Display_b()
    {
        Console.WriteLine("Переменная b = {0}", b);
    }
    protected void Display_e()
    {
        Console.WriteLine("Переменная e = {0}", e);
    }
}
```

Так как класс **State** объявлен с модификатором **public**, он будет доступен из любого места программы, а также из других программ и сборок. Класс **State** имеет пять полей для каждого уровня доступа и одну переменную без модификатора, которая является закрытой по умолчанию.

Также имеются четыре метода, которые будут выводить значения полей класса на экран. Обратите внимание, что поскольку все модификаторы позволяют использовать члены класса внутри данного класса, то и все переменные класса, включая закрытые, доступны всем его методам, так как все находятся в контексте класса **State**.

Рассмотрим на примере возможность использования переменных класса **State** в методе **Main** класса **Program**:

```
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State();
        // присвоить значение переменной a не получится,
        // так как она закрытая и класс Program её не видит
        // И данную строку среда подчеркнёт как неправильную
        state1.a = 4; //Ошибка, получить доступ нельзя
                      // то же самое относится и к переменной b
```

```
        state1.b = 3; // Ошибка, получить доступ нельзя
                      // присвоить значение переменной с не
получится,
                      // так как класс Program
                      // не является классом-наследником класса
State
        state1.c = 1; // Ошибка, получить доступ нельзя
                      // переменная d с модификатором internal
                      // доступна из любого места программы
                      // поэтому спокойно присваиваем ей значение
        state1.d = 5;
        // переменная e так же доступна
        // из любого места программы
        state1.e = 8;
        // переменная f общедоступна
        state1.f = 8;
        // Попробуем вывести значения переменных
    // Так как этот метод объявлен как private,
    // можно использовать его только внутри класса State
        state1.Display_f();
        // Ошибка, получить доступ нельзя
        // Так как этот метод объявлен как protected,
        // а класс Program не является
        // наследником класса State
        state1.Display_e();
        // Ошибка, получить доступ нельзя
        // Общедоступный метод
        state1.Display_a();

        // Метод доступен из любого места программы
        state1.Display_b();
        Console.ReadLine();
    }
}
```

Таким образом, установили только переменные **d**, **e** и **f**, так как их модификаторы позволяют использование в данном контексте. И оказались доступны только два метода: **state1.Display_a()** и **state1.Display_b()**. Однако, так как значения переменных **a** и **b** не были установлены, то эти методы выведут нули, так как значение переменных типа **int** по умолчанию инициализируются нулями.

Несмотря на то, что модификаторы **public** и **internal** похожи по своему действию, они имеют большое отличие. Классы и члены класса с модификатором **public** также будут доступны и другим программам, если данные класса поместить в динамическую библиотеку **dll** и потом использовать её в этих программах. Благодаря такой системе модификаторов

доступа можно скрывать некоторые моменты реализации класса от других частей программы. Такое сокрытие называется инкапсуляцией.

Кроме обычных методов в языке C# предусмотрены специальные методы доступа – свойства. Они обеспечивают простой доступ к полям класса и помогают узнать их значение или выполнить его установку.

Стандартное описание свойства имеет следующий синтаксис:

```
[модификатор_доступа] возвращаемый_тип произвольное_название
{
 // код свойства
}
```

Например:

```
class Person
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

В приведённом примере есть закрытое поле **name** и общедоступное свойство **Name**. Они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной **name**. Стандартное определение свойства содержит блоки **get** и **set**. Блок **get** – возвращает значение поля, а блок **set** – устанавливает. Параметр **value** представляет передаваемое значение.

Возможно также использовать данное свойство следующим образом:

```
Person p = new Person();
// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
p.Name = "Tom";
```

```
// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get
string personName = p.Name;
```

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Свойства позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Пусть надо установить проверку по возрасту:

```
class Person
{
    private int age;
    public int Age
    {
        set
        {
            if (value < 18)
            {
                Console.WriteLine("Возраст должен быть
               больше 18");
            }
            else
            {
                age = value;
            }
        }
        get { return age; }
    }
}
```

Блоки **set** и **get** не обязательно одновременно должны присутствовать в свойстве. Например, можно закрыть свойство от установки, чтобы только получать значение. Для этого опускаем блок **set**. И наоборот, можно удалить блок **get**, тогда можно будет только установить значение, но нельзя получить:

```
class Person
{
    private string name;
    // свойство только для чтения
    public string Name
    {
        get
        {
            return name;
        }
    }
    private int age;
```

```
    // свойство только для записи
    public int Age
    {
        set
        {
            age = value;
        }
    }
}
```

Модификаторы доступа можно применять не только ко всему свойству, но и к отдельным блокам – либо **get**, либо **set**. При этом можно применить модификатор только к одному из блоков:

```
class Person
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
        private set
        {
            name = value;
        }
    }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Теперь закрытый блок **set** можно будет использовать только в данном классе, но никак не в другом:

```
Person p = new Person("Tom", 24);
// Ошибка - set объявлен с модификатором private
//p.Name = "John";
Console.WriteLine(p.Name);
```

Через свойства устанавливается доступ к приватным переменным класса. Подобное сокрытие состояния класса от вмешательства извне представляет механизм инкапсуляции, который является одной из ключевых концепций объектно-ориентированного программирования. Применение модификаторов доступа типа **private** защищает переменную от внешнего доступа. Для управления доступом во многих языках программирования используются специальные методы, геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Например, есть некоторый класс **Account**, в котором определено поле **sum**, представляющее сумму:

```
class Account
{
    public int sum;
}
```

Поскольку переменная **sum** является публичной, то в любом месте программы мы можем получить к ней доступ и изменить её, в том числе установить какое-либо недопустимое значение, например отрицательное. Вряд ли подобное поведение является желательным. Поэтому применяется инкапсуляция для ограничения доступа к переменной **sum** и сокрытию её внутри класса:

```
class Account
{
    private int sum;
    public int Sum
    {
        get { return sum; }
        set
        {
            if (value > 0)
            {
                sum = value;
            }
        }
    }
}
```

Свойства управляют доступом к полям класса. Однако, если полей десять и более, то определять каждое и писать для него однотипное свойство было бы утомительно. Поэтому, начиная с версии **.NET 4.0**, в фреймворк были добавлены автоматические свойства, которые имеют сокращённое объявление:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

На самом деле тут также создаются поля для свойств, только их создаёт не программист в коде, а компилятор автоматически генерирует их при компиляции.

С одной стороны, автоматические свойства довольно удобны. С другой стороны, стандартные свойства имеют ряд преимуществ: например, они могут инкапсулировать дополнительную логику проверки значения; нельзя создать автоматическое свойство только для записи или чтения, как в случае со стандартными свойствами.

Ранее, для того чтобы использовать какой-нибудь класс и его методы, устанавливать и получать его поля, создавали его объект. Однако если данный класс имеет статические методы, то чтобы получить к ним доступ, необязательно создавать объект этого класса. Например, создадим новый класс **Algorithm** и добавим в него две функции для вычислений числа Фибоначчи и факториала:

```
class Algorithm
{
    public static double pi = 3.14;
    public static int Factorial(int x)
    {
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return x * Factorial(x - 1);
            16
        }
    }
    public static int Fibonachi(int x)
    {
        if (x == 0)
        {
            return 1;
        }
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return Fibonachi(x - 1) + Fibonachi(x - 2);
        }
    }
}
```

Ключевое слово **static** при определении переменной и методов указывает, что данные члены будут доступны для всего класса, т. е. будут статическими, используем их в программе:

```
int num1 = Algorithm.Factorial(5);
int num2 = Algorithm.Fibonachi(5);
Algorithm.pi = 3.14159;
```

При использовании статических членов класса необязательно создавать экземпляр класса, можно обратиться к ним напрямую.

Для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса. Например, если создать два объекта класса **Algorithm**, то оба этих объекта будут хранить ссылку на один участок в памяти, где хранится значение **pi**:

```
Algorithm algorithm1 = new Algorithm();
Algorithm algorithm2 = new Algorithm();
```

Нередко статические поля применяются для хранения счётчиков. Например, пусть есть класс **State**, и нужен счётчик, который позволял бы узнать, сколько объектов **State** создано:

```
class State
{
    private static int counter = 0;
    public State()
    {
        counter++;
    }
    public static void DisplayCounter()
    {
        Console.WriteLine("Создано {0} объектов State",
       counter);
    }
}
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State();
        State state2 = new State();
        State state3 = new State();
        State state4 = new State();
        State state5 = new State();
        State.DisplayCounter(); // 5
        Console.Read();
    }
}
```

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам, если таковые имеются:

```
class State
{
    static State()
    {
        Console.WriteLine("Создано первое государство");
    }
}
class Program
{
    static void Main(string[] args)
    {
        State s1 = new State(); // здесь сработает статический
конструктор
        State s2 = new State();

        Console.ReadLine();
    }
}
```

Наследование **inheritance** является одним из ключевых моментов ООП. Благодаря наследованию можно расширить функциональность уже существующих классов за счёт добавления нового функционала или изменения старого. Пусть есть класс **Person**, описывающий отдельного человека:

```
class Person
{
    private string _firstName;
    private string _lastName;

    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }
    public void Display()
    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}
```

Но вдруг потребовался класс, описывающий сотрудника предприятия – класс **Employee**. Поскольку этот класс будет реализовывать тот же функционал, что и класс **Person**, так как сотрудник также человек, то было бы рационально сделать класс **Employee** производным или наследником от класса **Person**, который, в свою очередь, называется базовым классом или родителем:

```
class Employee : Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса **Employee** базовым является **Person**, и поэтому класс **Employee** наследует все те же свойства, методы, поля, которые есть в классе **Person**.

Таким образом, наследование реализует отношение **is-a**, что значит «является»:

```
static void Main(string[] args)
{
    Person p = new Person
    {
        FirstName = "Bill",
        LastName = "Gates"
    };
    p.Display();
    p = new Employee
    {
        FirstName = "Denis",
        LastName = "Ritchi"
    };
    p.Display();
    Console.Read();
}
```

И поскольку объект класса **Employee** является также и объектом класса **Person**, то можно так определить переменную:

```
Person p = new Employee();
```

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

1. Не поддерживается множественное наследование, класс может наследоваться только от одного класса. Хотя проблема множественного наследования реализуется с помощью концепции интерфейсов.

2. При создании производного класса надо учитывать тип доступа к базовому классу – тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. Таким образом, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

3. Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

Вернёмся к классам **Person** и **Employee**. Хотя **Employee** наследует весь функционал от класса **Person**, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_firstName);
    }
}
```

Этот код не сработает и выдаст ошибку, так как переменная **_firstName** объявлена с модификатором **private** и поэтому доступ к ней имеет только класс **Person**. Но зато в классе **Person** определено общедоступное свойство **FirstName**, которое можно использовать, поэтому следующий код будет работать нормально:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(FirstName);
    }
}
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **public**, **internal**, **protected** и **protected internal**.

Теперь добавим в классы **Person** и **Employee** конструкторы:

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

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

Класс **Person** имеет стандартный конструктор, который устанавливает два свойства. Поскольку класс **Employee** наследует и устанавливает те же свойства, что и класс **Person**, то логично было бы не писать каждый раз код установки, а вызвать соответствующий код класса **Person**. К тому же свойств, которые надо установить, и параметров может быть гораздо больше.

С помощью ключевого слова **base** можно обратиться к базовому классу. В нашем случае в конструкторе класса **Employee** надо установить имя, фамилию и компанию. Имя и фамилия передаются на установку в конструктор базового класса **Person** с помощью выражения **base(fName, lName)**:

```
static void Main(string[] args)
{
    Person p = new Person("Bill", "Gates");
    p.Display();
    Employee emp = new Employee("Tom", "Simpson", "Microsoft");
    emp.Display();
    Console.Read();
}
```

Задание на лабораторную работу

Постройте иерархию классов в соответствии с вариантом задания:

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

Вариант	Задания
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
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. Спроектируйте абстракции и представьте иерархию классов в виде схемы.
2. Разработайте конструкторы, атрибуты и методы для каждого из определяемых классов.
3. Реализуйте программу на языке C# в соответствии с вариантом исполнения, используя экземпляры описанных классов.
4. Примените и объясните необходимость использования принципа инкапсуляции.

Контрольные вопросы

1. Что понимается под термином «класс»?
2. Какие элементы определяются в составе класса?
3. Каково соотношение понятий «класс» и «объект»?
4. Что понимается под термином «члены класса»?
5. Какие члены класса Вам известны?
6. Какие члены класса содержат код?
7. Какие члены класса содержат данные?
8. Перечислите пять разновидностей членов класса специфичных для языка C#.
9. Что понимается под термином «конструктор»?
10. Сколько конструкторов может содержать класс языка C#?
11. Приведите синтаксис описания класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
12. Какие модификаторы типа доступа Вам известны?
13. В чём заключаются особенности доступа членов класса с модификатором public?

14. В чём заключаются особенности доступа членов класса с модификатором private?

15. В чём заключаются особенности доступа членов класса с модификатором protected?

16. В чём заключаются особенности доступа членов класса с модификатором internal?

17. Какое ключевое слово языка C# используется при создании объекта?

18. Приведите синтаксис создания объекта в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

19. В чём состоит назначение конструктора?

20. Каждый ли класс языка C# имеет конструктор?

21. Какие умолчания для конструкторов приняты в языке C#?

22. Каким значением инициализируются по умолчанию переменные ссылочного типа?

23. В каком случае по умолчанию не используется конструктор класса?

24. Приведите синтаксис конструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

25. Что понимается под термином «деструктор»?

26. В чём состоит назначение деструктора?

27. Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

28. Что понимается под термином «наследование»?

29. Что общего имеет дочерний класс с родительским?

30. В чём состоит различие между дочерним и родительским классами?