Лабораторная работа №07. Классы и наследования в языке C#

1 Цель и порядок работы

Цель работы – познакомиться с основой объектного подхода в языке C#, созданием объектов, классов и механизмом наследования.

Порядок выполнения работы:
- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Наследование в C#

Наследование позволяет повторно использовать уже имеющиеся классы, но при этом расширять их возможности. Существует два вида наследования – наследование типа, при котором новый тип получает все свойства и методы родителя и наследование интерфейса, при котором новый тип получает от родителя сигнатуру методов, без их реализации. Все классы, для которых не указан базовый класс, наследуются от класса System.Object (краткая форма названия object). C# не поддерживает множественного наследования. Это означает, что класс в C# может быть наследником только одного класса, но при этом может реализовывать несколько интерфейсов. Другими словами, класс может наследоваться не более чем от одного базового класса и нескольких интерфейсов.

Спецификаторы доступа при наследовании

В C# существуют следующие спецификаторы доступа: private, protected, public, internal, protected internal.

private означает, что никакие другие классы не могут получить доступ к методу. Если в базовом классе объявлены private-методы, то наследник не может ими воспользоваться.

protected предоставляет доступ только для классов, которые наследуются от данного. Для остальных классов protected-методы недоступны.

public делает метод полностью открытым, то есть он является доступным для всех классов.

Ключевое слово internal является модификатором доступа для типов и членов типов. Внутренние типы или члены доступны только внутри файлов в одной и той же сборке.

При использовании ключевого слова protected internal методы и свойства доступны только в пределах одной сборки, а также для всех производных элементов.

Классам, как и их элементам, может быть назначен любой из этих уровней доступа. Если для элемента класса указан иной модификатор прав доступа, чем для класса,- приоритет у более строгого модификатора.

```
public class Human
{
  private String fName;
  public String FirstName
   {
    get { return fName; }
   set { fName = value; }
   }
}
```

Никакой другой класс не сможет получить доступ к полю fName (так как оно имеет модификатор доступа private), к свойству FirstName получат доступ все классы (потому что оно имеет модификатор доступа public).

Особенности использования конструкторов при наследовании

Конструктор наследования в C# имеет следующий вид:

```
class Наследуемый_класс : Базовый_класс
{
// поля, свойства, события и методы класса
}
```

Если класс наследуется от базового класса и нескольких интерфейсов, то они перечисляются через запятую:

```
class Наследуемый_класс : Базовый_класс,
                          Интерфейс1, Интерфейс2
{
// поля, свойства, события и методы класса
}
```

При создании класса – наследника на самом деле вызывается не один конструктор, а целая цепочка конструкторов. Сначала выбирается конструктор класса, экземпляр которого создается. Этот конструктор пытается обратиться к конструктору своего непосредственного базового класса. Этот конструктор в свою очередь пытается вызвать конструктор своего базового класса. Так происходит, пока не доходим до класса System.Object, который не имеет базового класса. В результате имеем последовательный вызов конструкторов всех классов

иерархии, начиная с System.Object заканчивая классом, экземпляр которого хотим создать. В этом процессе каждый конструктор инициализирует поля собственного класса.

Для каждого класса можно определить несколько конструкторов. Если мы для класса-наследника хотим вызвать конструктор базового класса, то необходимо использовать ключевое слово base().

```
public Наследуемый_класс() : base()
{
// поля, свойства, события и методы класса
}
```

Усовершенствуем класс Human добавив конструктор не принимающий параметров; конструктор принимающий в качестве параметров имя, отчество и фамилию; конструктор принимающий в качестве параметров имя, отчество, фамилию и дату рождения:

```
public class Human
  {
    protected String fName;
    public String FirstName
    {
      get { return fName; }
      set { fName = value; }
    }
    protected String mName;
    public String MiddleName
    {
      get { return mName; }
      set { mName = value; }
    }
    protected String lName;
    public String LastName
    {
      get { return lName; }
      set { lName = value; }
    }
    protected DateTime birthday;
    public DateTime Birthday
    {
```

```
        get { return birthday; }
        set { birthday = value; }
    }
    public Human() { }
    public Human(String FirstName, String MiddleName,
                                        String LastName)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
    }
    public Human(String FirstName, String MiddleName,
                                            String LastName,
    DateTime Birthday)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
        this.birthday = Birthday;
    }
    public void Work()
    {  // Do something   }
}
```

Тогда класс-наследник может быть реализован следующим образом:

```
public class Employee : Human
{
    public Employee()
        : base()
    {
        // Create Employee object
    }
    public Employee(String FirstName, String MiddleName,
                                            String LastName)
        : base(FirstName, MiddleName, LastName) { }
    public Employee(String FirstName, String MiddleName,
```

```
        String LastName, DateTime Birthday)
            : base(FirstName, MiddleName, LastName, Birthday) { }
    }
```

Сокрытие имен при наследовании

Сокрытие имен происходит, когда в базовом классе и в классе-наследнике объявлены методы с одинаковым именем. В такой ситуации метод базового класса скрывается, и программа, может работать не так, как предусматривал программист. В таких случаях необходимо воспользоваться модификатором new, который скажет компилятору о явном намерении скрыть метод базового класса и использовать метод, объявленный в классе наследнике. Например, пусть у нас есть класс Human, который имеет метод Work().

```
class Human
{
  public void Work()
   {  }
}
```

Объявим еще один класс Employee, который наследуется от класса Human и объявим в нем метод, который тоже назовем Work(). Чтобы в дальнейшем избежать путаницы, воспользуемся модификатором new:

```
public class Employee : Human
{
public new void Work()
   { }
}
```

Ключевое слово base

Ключевое слово base используется для доступа к членам базового класса из производного, для вызова метода базового класса при его переопределении в классе наследнике и при создании конструктора класса наследника, который должен вызвать конструктор класса родителя. Доступ к базовому классу разрешен только в конструкторе, методе экземпляра или методе доступа экземпляра.

Предположим, что необходимо вызвать в классе Employee метод Work() определенный в базовом классе Human:

```
public class Employee : Human
{
```

```
  public Employee() : base()
   {
      // Create Employee object
    }
  public override void Work()
    {
     base.Work();
    // Do something great
     }
}
```

Использование ключевого слова sealed (бесплодные классы)

Иногда возникают ситуации, когда необходимо запретить наследовать некоторый класс или переопределять некоторый метод. Для этого класс или метод нужно объявить как терминальный. Для этого используют ключевое слово sealed. Объявим терминальный класс:

```
public sealed class Tutor : Human { }
```

Никакие другие классы не могут наследовать класс Tutor. При попытке это сделать компилятор выдаст ошибку. При этом сам класс Tutor может быть наследован от другого класса.

Аналогично можно запретить переопределять свойства и методы базового класса. Если член базового класса имеет модификатор наследования virtual, тогда наследник может закрыть дальнейшее переопределение члена. Разрешим переопределение наследуемыми классами метода Work() класса Human.

```
class Human
{
  public virtual void Work()
   {
     // Do something
    }
}
```

Тогда класс Employee может переопределить метод Work() и закрыть возможность его переопределения для собственных наследников.

```
public class Employee : Human
{
    public sealed override void Work()
```

```
    {
     // Do something great
     }
}
```

При попытке откомпилировать следующий код, компилятор выдаст ошибку

```
public class Manager : Employee
{
  public override void Work()
   {
     // Try to do something unbelievable
    }
}
```

2.2 Виртуальные методы

Иногда необходимо изменить методы, которые наследуются от базового класса. Для того чтобы была возможность его изменить используют виртуальные методы. Объявив метод или свойство класса как виртуальные, вы тем самым позволяете классам наследникам переопределять данный метод или свойство. Для этого используется ключевое слово **virtual.** Оно записывается в заголовке метода базового класса, например:

```
virtual public void Passport() . . .
```

Особенность использования виртуальных методов состоит в следующем:

- Поля-члены и статические методы не могут быть объявлены как виртуальные.
- Применение виртуальных методов позволяет реализовывать механизм позднего связывания.
- На этапе компиляции строится только таблица виртуальных методов, а конкретный адрес метода, который будет вызван, определяется на этапе выполнения.

При вызове метода - члена класса действуют следующие правила:

- для виртуального метода вызывается метод, соответствующий типу объекта, на который имеется ссылка;
- для не виртуального метода вызывается метод, соответствующий типу самой ссылки.

При позднем связывании определение вызываемого метода происходит на этапе выполнения (а не при компиляции) в зависимости от типа объекта, для которого вызывается виртуальный метод.

Виртуальные методы необходимы, когда классу-наследнику нужно изменить некоторые методы, определенные в базовом классе. Виртуальные методы позволяют определить базовому классу методы, реализация которых есть общей для всех производных классов, и методы, которые можно переопределять. Это позволяет поддерживать динамический полиморфизм. Определения классов-наследников собственных методов становится более гибким, по-прежнему оставляя в силе требование согласующегося интерфейса.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово **override**, например:

override public void Passport() . . .

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Пример использования виртуальных методов:

```
class Monster
   {   string name; // закрытые поля
      int health, ammo;
      public Monster()
      {
         this.name = "Noname";
         this.health = 100; this.ammo = 100;
      }
      public Monster(string name) : this()
      { this.name = name; }
      public Monster(int health, int ammo, string name)
      {
         this.name = name;
         this.health = health;
         this.ammo = ammo;
      }
      public string GetName()
      { return name; }
      public int GetHealth()
      { return health; }
```

```
    public int GetAmmo()
    { return ammo; }
    virtual public void Passport()
    {
        Console.WriteLine("Monster {0} \t health = {1} ammo = {2} ",
                                    name, health, ammo);
    }
    public int Health // свойство Health связано с полем health
    {
        get { return health; }
        set { if (value > 0) health = value; else health = 0; }
    }
    public int Ammo // свойство Ammo связано с полем ammo
    {
        get { return ammo; }
        set { if (value > 0) ammo = value; else ammo = 0; }
    }
    public string Name // свойство Name связано с полем name
    { get { return name; } }
}
class Daemon : Monster
{   int brain; // закрытое поле
    public Daemon ()
    { brain = 1; }
    public Daemon (string name, int brain) : base(name) // 1
    { this.brain = brain; }
    public Daemon (int health, int ammo, string name, int brain)
        : base(health, ammo, name) // 2
    { this.brain = brain; }
    override public void Passport()
    {
        Console.WriteLine("Daemon {0} \t health = {1} ammo = {2}
                    brain = {3} ", Name, Health, Ammo, brain);
    }
}
```

```
class Class1
{
   static void Main()
   {
      const int n = 3;
      Monster[] stado = new Monster[n];
      stado[0] = new Monster("Monia");
      stado[1] = new Monster("Monk");
      stado[2] = new Daemon("Dimon", 3);
      foreach (Monster elem in stado) elem.Passport();
      for (int i = 0; i < n; ++i) stado[i].Ammo = 0;
      Console.WriteLine();
      foreach (Monster elem in stado) elem.Passport();
   }
}
```

Результат выполнения программы изображен на рисунке 2.1.

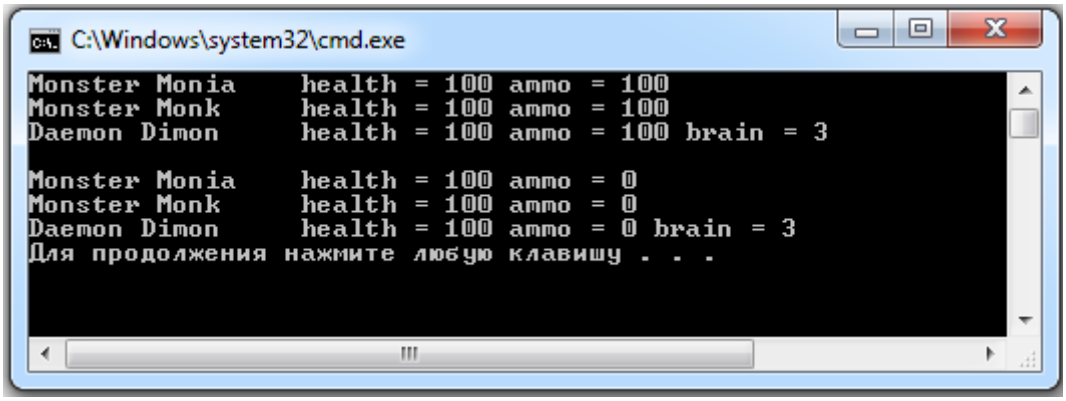

Рис. 2.1 Результат выполнения программы.

Теперь в циклах 1 и 3 вызывается метод Passport, соответствующий типу объекта, помещенного в массив.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков необязательно: если он выполняет устраивающие потомка действия, метод наследуется.

Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы (Virtual Method Table, VMT), из VMT выбирается адрес метода, а затем управление передается этому методу. Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.

При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

2.3 Абстрактные классы

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором **abstract**.

Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например:

```
abstract class Spirit
   {
    public abstract void Passport();
   }
 class Monster : Spirit
  { …
   override public void Passport()
    {
     Console.WriteLine( "Monster {0} \t health = {1} ammo = {2} " ,
                        name, health, ammo );
    }
   … }
class Daemon : Monster
   { …
override public void Passport()
      {
         Console.WriteLine("Daemon {0} \t health = {1} ammo = {2}
```

```
                brain = {3} ", Name, Health, Ammo, brain);
        }
    … }
```

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов. Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.

Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать полиморфные методы, работающие с объектом любого типа в пределах одной иерархии. Полиморфизм в различных формах является мощным и широко применяемым инструментом ООП.

2.4 Базовый класс Object

Корневой класс System.Object всей иерархии объектов .NET, называемый в C# object, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Класс object часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется полиморфизм.

Открытые методы класса System.Object:

Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

public virtual bool Equals(object obj);

Метод Equals с двумя параметрами возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool Equals(object ob1, object ob2);

Метод GetHashCode формирует хеш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод Equals, необходимо перегрузить и метод GetHashCode. Синтаксис:

public virtual int GetHashCode ();

Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип

Type. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения. Синтаксис:

public Type GetType ();

Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool ReferenceEquals (object ob1, object ob2);

Метод ToString по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис:

public virtual string ToString ()

В производных объектах эти методы часто переопределяют. Например, можно переопределить метод Equals для того, чтобы задать собственные критерии сравнения объектов, потому что часто бывает удобнее использовать для сравнения не ссылочную семантику (равенство ссылок), а значимую (равенство значений).

Пример применения и переопределения методов класса object.

```
class Monster
  {
    string name;
    int health, ammo;
    public Monster(int health, int ammo, string name)
    {
      this.health = health;
      this.ammo = ammo;
      this.name = name;
    }
    public override bool Equals(object obj)
    {
      if (obj == null || GetType() != obj.GetType()) return false;
      Monster temp = (Monster)obj;
      return health == temp.health &&
      ammo == temp.ammo &&
      name == temp.name;
    }
    public override int GetHashCode()
```

```
    { return name.GetHashCode(); }
    public override string ToString()
    {
        return string.Format("Monster {0} \t health = {1} ammo = { 2 } ",
                                                name, health, ammo);
    }
}
class Class1
{
    static void Main()
    {
        Monster X = new Monster(80, 80, "Вася");
        Monster Y = new Monster(80, 80, "Вася");
        Monster Z = X;
        if (X == Y) Console.WriteLine("X == Y ");
        else Console.WriteLine(" X != Y ");
        if (X == Z) Console.WriteLine(" X == Z ");
        else Console.WriteLine(" X != Z ");
        if (X.Equals(Y)) Console.WriteLine(" X Equals Y ");
        else Console.WriteLine(" X not Equals Y ");
        Console.WriteLine(X.GetType());
    }
}
```

Результат выполнения программы изображен на рисунке 2.2.

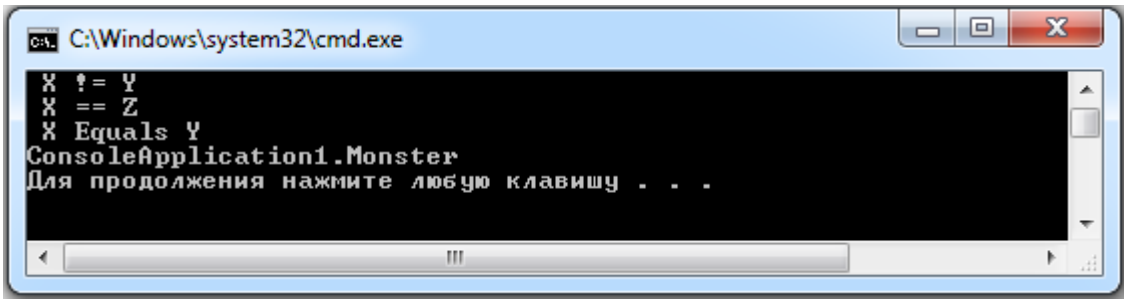

Рис. 2.2 Результат выполнения программы.

В методе Equals сначала проверяется переданный в него аргумент. Если он равен null или его тип не соответствует типу объекта, вызвавшего метод, возвращается значение false. Значение true формируется в случае попарного равенства всех полей объектов.

Метод GetHashCode просто делегирует свои функции соответствующему методу одного из полей. Метод ToString формирует форматированную строку, содержащую значения полей объекта.

Анализируя результат работы программы, можно увидеть, что в операции сравнения на равенство сравниваются ссылки, а в перегруженном методе Equals - значения.

2.5 Наследование исключений

В C# исключения представлены в виде класса. Программист имеет возможность воспользоваться встроенными исключениями, или создавать свои. Все исключения в C# наследуются от класса System.Exception. Из этого класса выведены два класса исключений: SystemException (исключения, которые генерируются общеязыковой средой выполнения CLR) и ApplicationException (генерируются прикладными программами).

При создании исключений в своих программах программист наследует их от ApplicationException. Классы-наследники должны иметь как минимум четыре конструктора: один по умолчанию, второй, задающий свойство сообщению, третий, задающий свойства Message и InnerException, четвертый – для сериализации исключения, поскольку новые классы исключений должны быть сериализируемые.

3 Контрольные вопросы

1. Что понимается под термином «наследование»?
2. Какая классификация объектов соответствует наследованию?
3. Что общего имеет дочерний класс с родительским?
4. В чем состоит различие между дочерним и родительским классами?
5. Приведите синтаксис описания наследования классов в общем виде.
6. Проиллюстрируйте его фрагментом программы на языке C#.
7. Какому отношению соответствует иерархия классов?
8. Какому отношению соответствует иерархия объектов?
9. Что понимается под термином «виртуальный метод»?
10. Какое ключевое слово языка C# используется для определения виртуального метода?
11. В чем состоит особенность виртуальных методов в производных (дочерних) классах?
12. В какой момент трансляции программы осуществляется выбор версии виртуального метода?
13. Какие условия определяют выбор версии виртуального метода?
14. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе?
15. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в производном (дочернем) классе?
16. Какие модификаторы недопустимы для определения виртуальных методов?
17. Что означает термин «переопределенный метод»?

18. В какой момент трансляции программы осуществляется выбор вызываемого переопределенного метода?
19. Приведите синтаксис виртуального метода в общем виде.
20. Что понимается под термином «абстрактный класс»?
21. В чем заключаются особенности абстрактных классов?
22. Какой модификатор языка C# используется при объявлении абстрактных методов?
23. Являются ли абстрактные методы виртуальными?
24. Используется ли модификатор virtual языка C# при объявлении абстрактных методов?
25. Возможно ли создание иерархии классов посредством абстрактного класса?
26. Возможно ли создание объектов абстрактного класса?

4 Задание

1. Выбрать задание согласно варианта.
2. Порядок выполнения работы:
   - разработать поля, методы и свойства для каждого из определяемых классов;
   - все поля классов должны быть описаны с ключевым словом private;
   - реализовать для каждого класса конструкторы по умолчанию и конструкторы с параметрами;
   - методы по изменению значения полей, поиска информации из массива данных объектов по определенным критериям.
3. Реализовать программу на C# в соответствии с вариантом исполнения.
4. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Построить иерархию классов в соответствии с вариантом 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, счет.
11. Автомобиль, поезд, транспортное средство, экспресс.
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
13. Республика, монархия, королевство, государство.
14. Млекопитающее, парнокопытное, птица, животное.
15. Корабль, пароход, парусник, корвет.
16. Самолет, автомобиль, корабль, транспортное средство.
17. Точка, линия, фигура плоская, фигура объемная.
18. Картина, рисунок, репродукция, пейзаж.
19. Статья, раздел, журнал, издательство.
20. Квартира, дом, улица, населенный пункт.