МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к лабораторным работам

«Объектно-ориентированное программирование на языке C#»

по дисциплине «Технологии программирования»

для студентов специальностей

122 – Компьютерные науки и информационные технологии,

124 – Системный анализ, в том числе для иностранных студентов

Утверждено
редакционно-издательским
советом университета,
протокол № 1 от 22.06.17 г

Харьков
НТУ «ХПИ»
2018

Методические указания к лабораторным работам «Объектно-ориентированное программирование на языке C#» по дисциплине «Технологии программирования» : для студентов специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ, в том числе для иностранных студентов /сост.: Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков. – Харьков : НТУ «ХПИ», 2018. – 80 с. – На рус. яз.

Составители: Ю. Н. Кожин
О. Н. Малых
В. Ф. Прокопенков

Рецензент А. В. Горелый

Кафедра системного анализа и информационно-аналитических технологий

ВСТУПЛЕНИЕ

Язык C# и технология программирования .NET Framework пришли на смену языку C/C++ и обычному программированию для Windows. Возможности, предлагаемые платформой .NET, позволяют радикально облегчить жизнь программистов и разрабатывать программные приложения разного назначения.

В отличие от других языков программирования язык C# является полностью объектно-ориентрованным. Это означает, что в программе, составляемой на этом языке, не может быть кода, определенного вне какого-либо класса.

Принципы объектно-ориентированного программирования разработаны и известны давно. Классическим примером объекто-ориентированного языка программирования является язык C++. Но реализация принципов объектно-ориентированного программирования в разных языках имеет свои особенности. Чтобы эффективно использовать возможности языка C# и средства платформы .NET для разработки производительных программ, необходимо в полном объеме овладеть принципами объектно-ориентированного программирования.

Предлагаемые методические указания помогут студентам ознакомиться с основными принципами реализации объектно-ориентированного программирования для платформы .NET Framework на языке C# и успешно их использовать для разработки совершенных программ.

Методические указания содержат необходимые теоретические сведения, а также рекомендации к выполнению лабораторных работ.

1. ИСПОЛЬЗОВАНИЕ КЛАССОВ В ЯЗЫКЕ C#

Объектно-ориентированное программирование в C# построено на классах. Класс может выполнять одну из двух ролей:

- модуля как архитектурной единицы программной системы;
- типа данных, для которого он задает реализацию определенной абстракции данных и связанную с ней функциональность.

Язык C# допускает использование классов как типов (классов, объявленных без спецификатора ***static***), так и модулей (классов, объявленных с использованием спецификатора ***static***).

1.1. Классы-модули

Использование классов-модулей обычно нацелено на реализацию только конкретной функциональности – классы-модули предоставляют свои функции-сервисы другим классам. К классам-модулям относятся, например, такие классы, как ***Console, Convert, Math*** и другие классы, объявленные со спецификатором ***static***.

Так, из перечисленных примеров классов-модулей класс ***Console*** обеспечивает сервис для работы с консолью, класс ***Convert*** обеспечивает возможности преобразования типа, а класс ***Math*** – различные вычислительные функции.

1.2. Классы-типы

Классы типы являются инструментом для моделирования объектов реального мира. Состав класса как типа данных определяется абстракцией данных и функциональностью, которую должен обеспечивать реализуемый им тип.

В такой роли класс – это отдельный программный модуль, поддерживающий определенную абстракцию данных и обеспечивающий необходимую функциональность реализуемого им типа данных.

Например, если необходимо реализовать тип комплексное число, то состав разрабатываемого класса типа должен обеспечивать как представление (хранение) как самого комплексного числа, так и операций над этим числом.

1.3. Синтаксис описания класса

В языке C# используется следующий синтаксис описания класса:

[атрибуты][модификаторы] ***class*** **имя_класса** [:список_родителей]

{

<тело_класса>

}

где:

class – зарезервированное слово;

имя_класса – идентификатор объявляемого класса;

[атрибуты] – атрибуты, с которыми объявляется класс;

[модификаторы] – модификаторы доступа объявляемого класса;

[:список_родителей] – список классов, от которых наследуется объявляемый класс;

<тело_класса> – содержит определение объявляемого класса, заключенное в фигурные скобки.

1.3.1 Атрибуты

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты

могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс *System.Attribute*, от которого образованы все остальные классы атрибутов. Следует понимать, что атрибуты представляют собой лишь средство донесения для компилятора некой дополнительной информации, которую он помещает в метаданные программного модуля.Библиотека классов .Net включает определение сотен атрибутов, которые можно использовать в своём коде. Например, при сериализации используются атрибуты *[Serializable]* и *[NonSerialized]*. Так, атрибут *[Serializable]* указывает классу *BinaryFormatter*, что объекты с данным атрибутом можно сохранять в бинарный файл. Если же атрибут для класса не указан, то сериализация к этому классу неприменима. Программист может определять свои настриваемые атрибуты, но как это делается – тема для отдельного изложения.

1.3.2 Модификаторы

В качестве *модификаторов* используются зарезервированные слова:

abstract – класс объявляется как абстрактный;

sealed – класс запрещается наследовать.

Модификаторы доступа (табл. 2.1) ***public, private, protected***, ***internal*** к классу, объявленному внутри другого класса, опредяляют, в какой области видимости программы могут быть созданы объекты класса.

Для класса, объявленного в пространстве имён, применим только модификатор доступа ***public.*** Класс, объявленный без модификатора доступа, по умолчанию считается заданным с модификатором ***internal*** (доступность классам одного проекта).

Таблица 2.1 – Модификаторы доступа класса

Модификатор	Уровень доступа к классу
public	Доступ возможен из любой сборки.
private	Доступ возможен только из кода в том же классе.
internal	Доступ возможен из кода той же сборки, где определён этот класс, но не из другой сборки.
protected	Доступ возможен только из кода в том же классе либо в классе, производном от этого класса.
protected internal	Доступ возможен только из кода в том же классе либо в классе, производном от этого класса, а также из кода той же сборки, где определён этот класс, но не из другой сборки.

1.3.3 Члены класса

В теле класса могут быть объявлены такие элементы класса:

- данные;
- конструкторы и деструкторы;
- методы;
- события;
- вложенные классы (структуры, делегаты, интерфейсы, перечисления).

Замечание: Вложенный класс следует использовать, когда этот класс носит вспомогательный характер и есть полная уверенность, что он никому не понадобится, кроме класса, в который он вложен и, возможно, его потомков. Внутренние классы обычно имеют модификатор доступа private или protected.

2. КЛАСС КАК ТИП ДАННЫХ

Данные-члены класса определяют состояние объектов класса, методы-члены (функции) – способы преобразования состояния объектов класса, определяя тем самым их поведение. Все объекты одного класса имеют один и тот же набор методов.

Данные и методы могут быть статическими (объявленные со спецификатором ***static***) и нестатическими членами класса (без спецификатора ***static***).

Статические члены принадлежат классу и используются без создания объектов класса. *Нестатические члены* являются собственностью каждого объекта класса.

Каждый член класса (данные или метод) имеет модификатор доступа, принимающий одно из четырех значений: ***public***, ***private***, ***protected***, ***internal*** (табл.2.2). Модификатором доступа по умолчанию является модификатор ***private***. Независимо от значения модификатора доступа, все члены класса доступны в области видимости метода этого класса.

2.1. Конструктор класса

Конструктор класса – это специальный метод, который вызывается для инициализации (определения начального состояния) созданного объекта при выполнении оператора ***new***. Для класса можно определить любое количество конструкторов, которые различаются сигнатурой. Конструкторы могут быть статическими и нестатическими. Статические конструкторы используются для инициализации статических дан-

ных-членов, нестатические – для инициализации данных-членов объектов.

Таблица 2.1 – Модификаторы доступа для членов класса

Модификатор	Уровень доступа к членам
public	Доступны из любого места приложения.
private	Доступны только членам этого же типа.
internal	Доступны из любого типа данной сборки, но не из внешних сборок.
protected	Доступны только членам этого же типа и типов, производных от него.
protected internal	Доступны любым типам данной сборки, а также типам, производным от типа, которому принадлежит данный член.

Если программист при объявлении класса не определит ни одного конструктора, то класс автоматически снабжается конструктором по умолчанию без параметров. Такой конструктор присвоит всем переменным-членам класса безопасные значения по умолчанию.

Синтаксис объявления конструктора класса имеет особенности:

- имя конструктора фиксировано и совпадает с именем класса,
- для конструктора не задается возвращаемое значение.

Обычно конструктор имеет модификатор доступа *public*.

Например:

```
public class Complex
    {
        double real=11;
        double image=22;
```

```
        public Complex(double real, double image)
        {
            this.real = real;
            this.image = image;
        }
    }
```

Программист должен понимать, как происходит создание объекта класса. В первом приближении алгоритм создания объекта класса можно описать так:

1) В стеке выделяется память под объявляемую переменную класса (ссылочная переменная), в которую записывается значение *null*.

2) В динамической памяти создается объект класса – выделяется память под объект.

3) Затем данные члены объекта инициализируются значениями по умолчанию: ссылочные поля – значением *null*, арифметические - нулями, строковые - пустой строкой. Эту работу выполняет конструктор по умолчанию, который, можно считать, всегда вызывается в начале процесса создания.

4) Если данные-члены класса объявлены с инициализацией (как в примере класса *Complex*), то выполняется инициализация полей объекта заданными значениями инициализации.

5) Затем выполняется заданный в операторе ***new*** конструктор с аргументами, тип которого определяется типами фактических параметров.

6) После создания и инициализации объекта в ссылочную переменную, определяющую создаваемый объект, записывается адрес созданного объекта в динамической памяти.

Убедиться в сказанном можно, если в режиме отладки просмотреть состояния создаваемого объекта. Например, при выполнении операторов:

```
Complex comp_val;
comp_val= new Complex(10, 20);
```

Реально, из-за наследования процесс создания объектов происходит сложнее, так как данные-члены объекта, наследуемые от родителя, создаются конструктором родителя, который и должен быть вызван в первую очередь.

При определении конструктора или других методов класса можно использовать зарезервированное слово ***this***, которое используется для ссылки на текущий экземпляр объекта. Использование указателя *this* позволяет избежать конфликтов между именами принимаемых параметров и именами внутренних переменных – членов класса (также такого конфликта можно избежать, определяя для аргументов методов имена, не совпадающие с именами данных-членов класса).

Также указатель *this* можно использовать для вызова конструктора. Если в класс *Complex* добавить такой конструктор (без параметров):

```
public Complex():this(0,0)
{
    …
}
```

то запись *this(0,0)* означает вызов конструктора этого же класса с двумя параметрами.

Но необходимо помнить, что через указатель ***this*** невозможно обратиться к статическим данным-членам и статическим методам-членам, так как они не принадлежат объектам класса, а принадлежат всему классу в целом.

2.2. Деструктор класса

Задача удаления ненужных объектов в .Net снята с программиста и возложена на соответствующий инструментарий – ***сборщик мусора***. В классическом варианте деструктор служит для удаления объектов и освобождения ресурсов, занятых объектом, в первую очередь оперативной памяти.

В языке C# у класса может быть деструктор, но он не занимается удалением объектов и не вызывается нормальным образом в ходе выполнения программы. Деструктор класса, если он есть, вызывается автоматически в процессе сборки мусора. Его функция – освобождение других ресурсов, используемых объектом.

Синтаксически деструктор определяется так:

```
~ имя_класса ( )
 {
   <тело деструктора>
 }
```

Имя деструктора строится из имени класса с предшествующим ему символом ~ (тильда). Как и у статического конструктора, у деструктора не указывается модификатор доступа.

2.3. Пример класса

Использования класса типа проиллюстрируем на примере типа комплексного числа.

```
using System;
namespace ConsoleApplication1
{
  public class Complex
  {
      public double real;
      public double image;

      public Complex()
      {
          real=default(double);
          image=default(double);
      }
      public Complex(double real, double image)
      {
          this.real = real;
          this.image = image;
      }
      public static Complex Add(Complex op1,
                                Complex op2)
      {
          return new Complex(op1.real + op2.real,
                             op1.image + op2.image);
      }
      public Complex AddTo(Complex op)
      {
          return new Complex(this.real + op.real,
                             this.image + op.image);
      }
  }
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.Title="Complex";
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.Black;
        Console.Clear();

        Complex c1 = new Complex();
        Complex c2 = new Complex(1, 2);
        Complex c3 = new Complex(5, 7);

        Console.WriteLine("c1={0}+j{1}", c1.real,
                          c1.image);
        Console.WriteLine("c2={0}+j{1}", c2.real,
                          c2.image);
        Console.WriteLine("c3={0}+j{1}", c3.real,
                          c3.image);

        Complex c4=Complex.Add(c2,c3);
        Complex c5=c2.AddTo(c3);

        Console.WriteLine("c4={0}+j{1}", c4.real,
                          c4.image);
        Console.WriteLine("c5={0}+j{1}", c5.real,
                          c5.image);
    }
}
```

На рис.2.1 представлен результат работы программы.

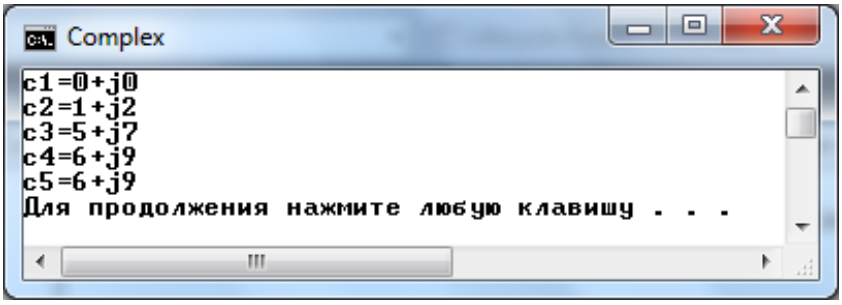

Рис.2.1 Результат выполнения примера

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ В ЯЗЫКЕ C#

В C# реализованы все три важнейших принципа объектно-ориентированного программирования:

- инкапсуляция – определяет, как объекты прячут свое внутреннее устройство;
- наследование – определяет, как поддерживается повторное использование кода;
- полиморфизм – определяет, как реализована поддержка выполнения нужного действия в зависимости от типа объекта.

3.1. Инкапсуляция

Инкапсуляцией называется способность прятать детали реализации объектов от пользователей этих объектов и проявляется:

- на уровне метода объекта;
- на уровне объекта в целом.

3.1.1 Инкапсуляция на уровне метода

Инкапсуляция на уровне метода означает, что пользователю метода не раскрывается, каким образом метод реализует свою функциональность. Если вы хотите использовать метод, все, что от вас требуется, – это вызвать этот метод и правильно передать ему необходимые параметры.

3.1.2 Инкапсуляция на уровне объекта

Инкапсуляция на уровне объекта в целом означает сокрытие от пользователя всех внутренних данных объекта (т.е. данных членов).

Если необходимо обратиться к сокрытым членам, то это можно сделать только применением открытых функций членов. В C# инкапсуляция на уровне объекта реализуется синтаксисом при помощи ключевых слов ***public***, ***private***, ***protected*** и ***internal***.

3.2. Поля

Применение спецификатора *public* при объявлении данных-членов класса обуславливает их открытость. Такие данные являются членами класса, но в полном смысле не инкапсулированы в нём, так как открыты для доступа извне класса. Обращение к таким данным возможно непосредственно по имени:

```
using System;
using System.Text;
namespace ConsoleApplication1
{
 class Encapsulate
 {
     public static int   statval;
     public      int val;

     public static void Main()
     {
         Cosole.WriteLine("statval={0}",
                          Encapsulate.statval);
         Encapsulate ob=new Encapsulate();
         Console.WriteLine("ob val={0}",ob.val);
     }
 }
}
```

Для статических данных используется форма обращения:

имя_класса.имя_данного_члена,

для объектных – форма обращения:

имя_объекта.имя_данного_члена .

Для данных-членов класса, объявленных со спецификатором *public*, применяется термин ***поле***. Недостаток использования полей состоит в том, что их открытость для пользователя приводит к их бесконтрольному использованию по отношению к классу или объекту. Например, если по логике класса полю можно присваивать значения из определенного диапазона, то это ограничение невозможно проконтролировать.

Для решения описанной проблемы можно использовать следующие решения:

1) использовать поля, доступные только для чтения (псевдоинкапсуляция);

2) закрыть данные от внешнего изменения, а для доступа к ним использовать методы чтения и записи;

3) закрыть данные от внешнего изменения, а для доступа к ним определить и использовать именованные свойства (как интерфейсные поля).

3.3. Поля, доступные только для чтения

Для объявления полей, доступных только для чтения, используется спецификатор ***readonly***. Такие поля могут быть как статическими, так и обычными. Любая попытка изменить значение такого поля приведет к ошибке компилятора:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
```

```
class Encapsulate
{
    public static readonly int    statval;
    public          readonly int  val;

    public static void Main()
    {
        Encapsulate.statval=1;    // ошибка
       Cosole.WriteLine("statval={0}",
                        Encapsulate.statval);
        Encapsulate  ob=new Encapsulate();
        ob.val=2;             // ошибка
        Console.WriteLine("ob val={0}",ob.val);
    }
}
}
```

При этом необходимо отметить, что и такой пример вызовет ошибки:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
 class Encapsulate
 {
     public static readonly int    statval;
     public    readonly int   val;

     public void Change(int statval,int val)
     {
         Encapsulate.statval=statval;  // ошибка
         this.val=val;                 // ошибка
     }

 }
}
```

Дело в том, что поля, объявленные как доступные только для чтения, не допускают возможности изменения их состояния, кроме начальной инициализации.

Нестатическое поле, объявленное только для чтения, можно инициализировать только при объявлении переменной.

Статическое поле, объявленное только для чтения, можно инициализировать только в специальном статическом конструкторе. При этом статический конструктор объявляется без модификатора доступа (считается *public*) и не допускает аргументов (конструктор по умолчанию). Пример:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
 class Encapsulate
 {
     public static readonly int    statval;
                        // инициализация при объявлении
     public     readonly int   val=111;
     static Encapsulate() //модификаторы области види-
                          //мости не указывать
     {                    //без параметров
         statval=999;
     }
 }
 class App
 {// Статический конструктор явно не вызывается
     public static void Main()
     {
       Encapsulate    ob=new Encapsulate();
       Console.WriteLine("statval={0}",
                          Encapsulate.statval);
```

```
            Console.WriteLine("ob val={0}",ob.val);
            Console.Read();
        }
    }
}
```

Результат выполнения:

statval=999

ob_val=111.

Если проанализировать пример, получается, что защита полей от записи превращает их из переменных в константы, допускающие только начальную инициализацию. Единственное отличием ***readonly***-полей от констант состоит в том, что значения ***readonly***-полям присваиваются во время создания объекта, а константам – на этапе компиляции.

3.4. Закрытые данные-члены

Второй способ реализации инкапсуляции – это закрытие данных и организация методов, доступных пользователю, для работы с ними.

Пример:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
 class Encapsulate
 {
   //закрытие данных от внешнего доступа
     private static int statval;
     private       int val;

     // методы для доступа к закрытым данным-членам
```

```
        public static void SetStatVal(int statval)
        {
            Encapsulate.statval=statval;
        }

        public static int  GetStatVal()
        {return Encapsulate.statval;}

        public void SetVal(int val){this.val=val;}

        public int  GetVal(){return this.val;}
    }

    class App
    {
        public static void Main()
        {
            Encapsulate  ob=new Encapsulate();

            Encapsulate.SetStatVal(888);
            ob.SetVal(222);

            Cosole.WriteLine("statval={0}",
                             Encapsulate.GetStatVal());
            Console.WriteLine("ob val={0}", ob.GetVal());
        }
    }
    }
```

3.5. Свойства класса

Кроме рассмотренных традиционных методов доступа для обращения к закрытым членам класса можно также использовать ***свойства*** (properties).

Свойство можно рассматривать как интерфейсный элемент, который по типу совпадает с данным-членом класса, который он представляет, но им не является.

Синтаксис обращения к свойству такой же, как и при обращении к полю, при этом при обращении к свойству на чтение вызывается специальный метод свойства *Get( )*, а при обращении на запись – метод *Set( )* этого свойства. Но явно указанные методы не вызываются – пользователь работает со свойством как с открытой переменной-членом данного класса. Пример:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
 class Encapsulate
 {
     private static   int statval;
     private      int val;

     // свойство для statval
     public static int StatVal
     {
         get {return statval;}
         set
         {
             if(value>=0 && value<=999)
               statval=value;
         }
     }
     // свойство для val
     public int Val
     {
         get {return val;}
         set
         {
             if(value>=0 && value<=999)
                 val=value;
         }
     }
 }
```

```
class App
{
     public static void Main()
     {
          Encapsulate  ob=new Encapsulate();

          Encapsulate.StatVal=888;
          ob.Val=222;

          Console.WriteLine("statval={0}",
                              Encapsulate.StatVal);
          Console.WriteLine("ob val={0}",ob.Val);
     }
}
}
```

Необходимо отметить, что для каждого свойства класса компилятором создается метод для чтения и записи соответствующего данного-члена с именем, формируемого по правилу: **get_имяСвойства** и **set_имяСвойства**. В этом можно убедиться, если изучить состав класса, используя утилиту дизасемблирования ildasm.exe (см. рис.3.1).

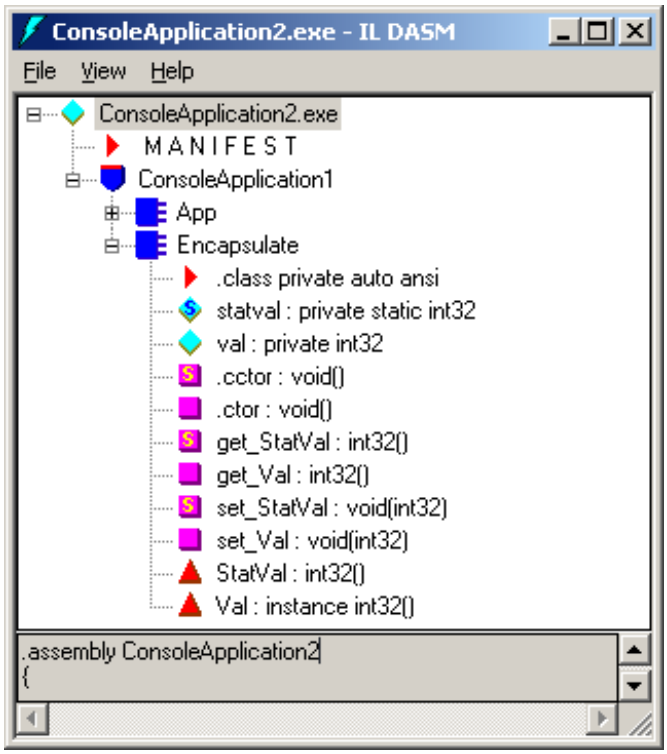

Рис.3.1. Просмотр класса в ildasm.exe

Свойства могут быть объявлены с ограничением доступа: то есть могут быть доступны только на чтение или только на запись. Это реализуется удалением из тела описания свойства метода, который реализует запрещаемое действие, как показано в примере:

```
class Encapsulate
{
    private static   int statval;
    private      int val;

    // свойство для statval – отсутствует метод get
    public static int StatVal
    {
        set
        {
            if(value>=0 && value<=999)
              statval=value;
        }
    }

    // свойство для val – отсутствует метод set
    public int Val
    {
        get {return val;}
    }
}
```

4. НАСЛЕДОВАНИЕ

Под *наследованием* понимается возможность создавать новые определения классов на основе существующих. Главная задача наследования – обеспечить повторное использование кода. Наследование позволяет расширять возможности, унаследованные от базового класса, в собственном производном классе.

Существует два основных вида наследования:

- классическое наследование, реализующее между классами отношение «быть» – is-a;
- и включение-делегирование, реализующее между классами отношение «иметь» – has-a.

4.1. Классическое наследование (is a)

Основная идея классического наследования состоит в том, что производные классы получают функциональность от базового класса-предка и дополняют ее новыми возможностями. Рассмотрим пример программы с двумя классами *Person* и *Worker*, которые реализуют классическое наследование:

```
using System;
using System.Text;

namespace ConsoleApplication3
{
  class Person
  {
    private string FullName;
    public int Age;

    public Person(string FullName, int Age)
    {
      this.FullName = FullName;
```

```
    this.Age = Age;
  }

  public string GetPerson()
  {
    StringBuilder s =
           new System.Text.StringBuilder("");
    s.AppendFormat("Name: {0} Age: {1} ", FullName,
                                          Age);
    return s.ToString();
  }
}

class Worker : Person
{
  private string Speciality;
  private int Stage;

  public Worker(string FullName,
                int Age,
                string Speciality,
                int Stage):base(FullName, Age)
  {
    this.Speciality = Speciality;
    this.Stage = Stage;
  }

  public string GetWorker()
  {
    StringBuilder s =
       new System.Text.StringBuilder(GetPerson());
    s.AppendFormat("Speciality: {0} Stage: {1}",
                   Speciality, Stage);
                   return s.ToString();
    }
  }

class App
{
  public static void Main()
  {
```

```
            Console.Title = "IS_A";
            Console.BackgroundColor = ConsoleColor.White;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.Clear();

            Person p = new Person("Person", 20);
            Worker w = new Worker("Worker", 25,
                                  "swipper", 5);
            Console.WriteLine(p.GetPerson());
            Console.WriteLine(w.GetWorker());
        }
    }
}
```

Результат выполнения программы представлен на рис.4.1.

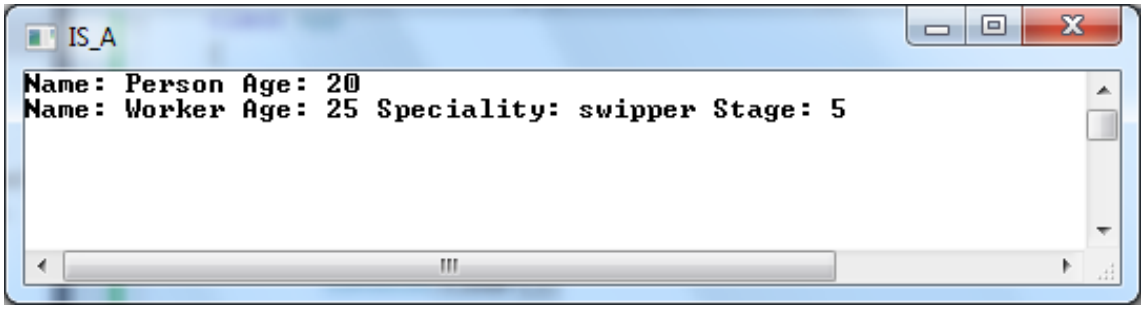

Рис. 4.1. Результат выполнения программы

В примере класс *Worker* наследуется от класса *Person* и представляет объекты персоны. Класс *Worker* используется для описания объектов работников и наследуется от класса *Person*.

В модели классического наследования базовые классы (в нашем случае *Person*) используются для того, чтобы определить характеристики, которые будут общими для всех производных классов (это элементы ***FullName, Age*** и метод ***GetPerson()***).

Производные классы (в нашем случае *Worker*) расширяют унаследованную функциональность за счет своих собственных, специфических членов (это элементы ***Speciality,Stage*** и метод ***GetWorker()***).

Класс *Worker* «имеет быть» (is a) классом *Person* + его особенности. В таком отношении наследования производные классы наследуют все члены базового класса. В производном классе обеспечивается доступ только к открытым членам базового класса, объявленным со спецификатором *public*. К закрытым членам базового класса, объявленным со спецификатором *private*, доступа нет. Для того, чтобы члены базового класса одновременно были и закрытыми, и доступными в производном классе они должны быть объявлены как ***protected***.

При определении конструктора производного класса для вызова конструктора базового класса используется ключевое слово ***base***, за которым в круглых скобках указываются параметры конструктора базового класса.

Слово ***base*** необходимо рассматривать как указатель на базовый класс, и его удобно использовать, когда необходимо обратиться к открытым или защищенным членам базового класса.

Реализация объектно-ориентированного программирования в языке C# имеет свои особенности:

1) наследование разрешено только от одного базового класса, что устраняет проблемы множественного наследования.

2) множественное наследование разрешено только для интерфейсов, для которых проблемы множественного наследовании решаются проще.

3) при определении класса имеется возможность запретить дальнейшее наследование от него. Для этого используется ключевое слово *sealed* (закрытый на ключ) и следующий синтаксис объявления класса:

public sealed class *Administrator* : *Person*

{

<тело класса>

}

4.2. Модель наследования включение-делегирование (has a)

Рассмотрим суть этой модели на примере. Пусть нам необходимо разработать класс, который в своей функциональности должен реализовать функциональность класса, который уже реализован (например, реализованный класс радиоприёмник *Radio*, а новый класс автомобиль *Car*). Можно было новый класс произвести от реализованного, но новый класс и имеющийся никакого естественного родства в плане своей функциональности не имеют.

В данном случае, новый класс будет строиться из имеющегося класса на основе модели включение-делегирование. Смысл такой модели в том, что новый класс будет содержать объект имеющегося класса, который будет делегировать новому классу свои функции, доступные через этот объект:

```
using System;
using System.Text;
namespace ConsoleApplication4
{
  class App
  {
    class Radio
    {
      public void TurnOn()
      {
        Console.WriteLine("Playing");
      }
    }
```

```
    class Car
    {
      private Radio radioset = new Radio();

      public void SetRadioOn()
      {
        radioset.TurnOn();
      }
    }

    public static void Main()
    {
      Console.Title = "HAS a";
      Console.BackgroundColor = ConsoleColor.White;
      Console.ForegroundColor = ConsoleColor.Black;
      Console.Clear();

      Car carmobile = new Car();

      carmobile.SetRadioOn();
    }
  }
  }
```

Результат выполнения примера программы представлен на рис.4.2:

Рис. 4.2. Результат выполнения программы

5. ВЛОЖЕННЫЕ КЛАССЫ

В C# можно определить один класс (тип) непосредственно внутри другого класса:

```
public class MyClass
{
 // члены внешнего класса
 …
  public class MyNestedClass
  {
        // члены внутреннего класса
  }
}
```

Такие типы называются вложенными (nested):

Как правило, вложенные типы – это исключительно вспомогательные типы, используемые только теми внешними типами, в которых они непосредственно определены. Обращение к вложенным типам напрямую из внешнего мира невозможно.

Реально функциональность вложенных типов практически совпадает с функциональностью внутренних типов в модели включения-делегирования, за исключением того, что для вложенных типов обеспечивается более строгий контроль области видимости. В этом отношении применение вложенных типов помогает обеспечить еще более полное соответствие принципам инкапсуляции. В такой технике предыдущий пример можно представить так:

```
using System;
using System.Text;
namespace ConsoleApplication5
{
  class App
  {
    class Car
    {
      public class Radio
      {
        public void TurnOn()
        {
          Console.WriteLine("Playing");
        }
      }

      private Radio radioset = new Radio();

      public void SetRadioOn()
      {
        radioset.TurnOn();
      }
    }

    public static void Main()
    {
      Console.Title = "HAS a";
      Console.BackgroundColor = ConsoleColor.White;
      Console.ForegroundColor = ConsoleColor.Black;
      Console.Clear();

      Car carmobile = new Car();

      carmobile.SetRadioOn();

      Radio  rad; // Ошибка - класс не доступен
    }
  }
}
```

6. ПОЛИМОРФИЗМ

Полиморфизм реализуется через наследование, определяется заложенными в язык возможностями и означает характер различного выполнения одноименных функций в классах, связанных наследованием. Существует две основные разновидности полиморфизма: классический полиморфизм и полиморфизм «для конкретного случая» (ad hoc).

6.1. Полиморфизм для конкретного случая

Полиморфизм для конкретного случая реализуется в других языках(в Basic, в C++), но не в C# и предполагает следующую модель:

1) пусть имеются классы, не связанные наследованием, которые имеют одноименную функцию;

2) свяжем с каждым элементом массива указателей на объекты типа *void** объект конкретного класса из рассматриваемых классов;

3) вызывая для каждого элемента массива одноименную функцию, мы по сути вызовем функцию того объекта, указатель на который хранит элемент массива.

Описанную ситуацию называют полиморфизмом конкретного случая, который реализуется технологией позднего связывания, когда тип объекта выясняется и становится ясен в процессе выполнения программы.

Полиморфизм для конкретного случая в C# не реализуется о чём свидетельствует следующий пример:

```
using System;
using System.Text;

namespace ConsoleApplication5
{

 class Class1
 {
   public void F()
   {
     Console.WriteLine("F_Class1()");
   }
 }

class Class2
 {
  public void F()
  {
    Console.WriteLine("F_Class2()");
  }
 }

 class App
 {
  public static void Main()
  {
   Console.Title = "AD HOC";
   Console.BackgroundColor = ConsoleColor.White;
   Console.ForegroundColor = ConsoleColor.Black;
   Console.Clear();

   Class1     c1 =new Class1();
   Class2     c2 =new Class2();

   c1.F();
   c2.F();

   Objectob;

   ob=c1;
   ob.F();
```

```
    ob=c2;
    ob.F();
   }
 }
}
```

Приведенный пример приводит к ошибке трансляции (см. рис.6.1).

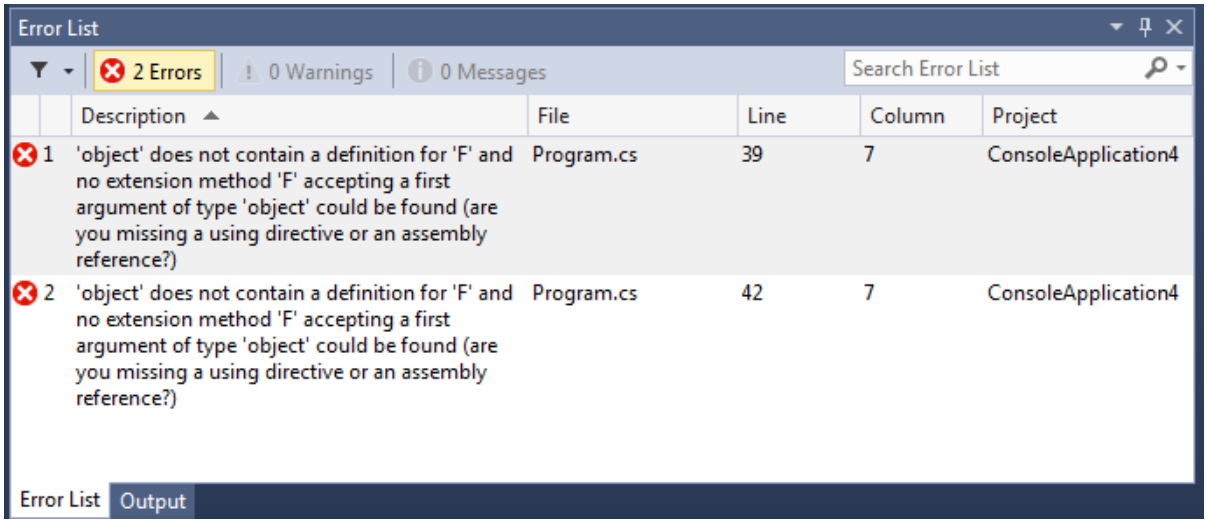

Рис.6.1. Ошибки при моделировании наследования «для конкретного случая»

Почему, такая модель наследования нереализуема в языке C#?

Ответ простой, потому что в отличие от типа *void** в языке C++ тип *object* в C# не является нетипизированным (а является конкретным типом). Поэтому попытка в примере выполнить оператор *ob.F()* логично приводит к ошибке – «*object*» не содержит определения для *F()*…

6.2. Классический полиморфизм

Классический полиморфизм основан на такой модели:

1) пусть имеется базовый класс и производный от него;
2) если в базовом классе определена функция, которая замеща-

ется в производном классе, то по отношению к этой функции рассматриваемые классы будут обладать полиморфизмом. То есть вызов одной и той же функции для объектов этих классов повлечет их разное функциональное поведение.

В языке C# возможны описанные ниже варианты, которые могут возникнуть при наследовании:

6.2.1. Простое наследование

Если в базовом классе определена функция, и мы производим от этого класса новый класс, в котором эта функция никак не переопределяется и если эта функция объявлена как *public* или *protected*, то она наследуется производным классом и будет доступна из него:

```
using System;
using System.Text;

namespace ConsoleApplication1
{
    class Base
    {
      public void F()
      {
        Console.WriteLine("F_Base()");
      }
    }

    class FromBase : Base
    {
      public void F2()
      {
        Console.WriteLine("F2()");
      }
    }

    class App
```

```
{
  public static void Main()
  {
    Console.Title = "simple inheritance";
    Console.BackgroundColor =
                              ConsoleColor.White;
    Console.ForegroundColor =
                              ConsoleColor.Black;
    Console.Clear();

    Base b = new Base();
    FromBase fb = new FromBase();

    b.F();
    fb.F();

    Base b2 = fb;
    b2.F();
  }
}
```

Результат выполнения программы (рис.6.2) подтверждае сказанное.

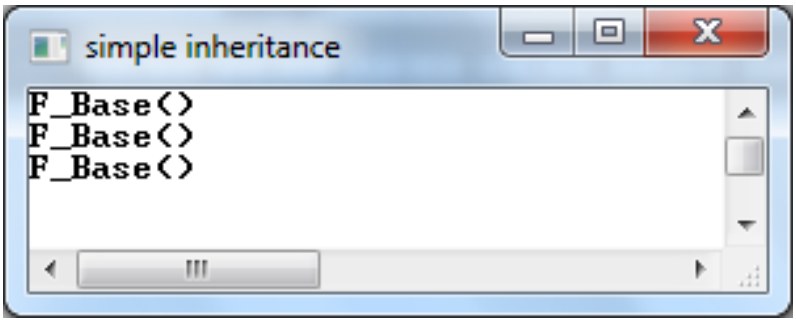

Рис.6.2. Простое наследование

6.2.2. Замещение функции базового класса

Замещение функции базового класса в производном классе состоит в том, что:

1) в базовом классе функция определяется со спецификатором ***virtual*** (или без него – это не принципиально),

2) в производном классе одноименная функция объявляется со спецификатором ***new*** или без него.

Это приводит к тому, что для объекта производного класса будут доступны две функции: функция производного класса, если мы обращаемся к объекту этого класса, и функция базового класса.

Возможность доступа к одноименной функции базового класса реализуется через ссылку ***base*** или приведением типа объекта к типу базового класса:

```
using System;
using System.Text;
namespace ConsoleApplication6
{

 class Base
 {
     public void F()
     {
       Console.WriteLine("F_Base()");
     }
 }

 class FromBase : Base
 {
     public new void F()
     {
       Console.WriteLine("F_FromBase()");
     }
 }

 class App
 {
     public static void Main()
     {
       Console.Title = "new";
       Console.BackgroundColor = ConsoleColor.White;
       Console.ForegroundColor = ConsoleColor.Black;
```

```
            Console.Clear();

            Base        b =new Base();
            FromBase    fb=new FromBase();

            b.F();
              fb.F();

              Base      b2=fb;
              b2.F();
        }
 }
}
```

Результат выполнения программы представлен на рис.6.3.

Рис.6.3. Замещение одноименной функции при наследовании

По исполнению можно заметить, что для базового класса вызывается ее функция *F( )*, а для производного – функция *F( )* производного класса.

При этом, если мы присвоили ссылку на объект производного класса переменной ссылке на базовый класс, то вызов *F( )* приводит к вызову функции базового класса.

Таким образом, если мы сознательно решили закрыть функцию базового класса, то при ее объявлении в производном классе мы должны указать спецификатор ***new***.

6.2.3 Реализация классического полиморфизма

Для реализации классического полиморфизма одноименная функция в базовом классе объявляется со спецификатором ***virtual,*** а в производном классе – со спецификатором ***override***.

Это приводит к тому, что функция базового класса аннулируется и для объекта производного класса будет доступна функция производного класса, что и представляется как ***полиморфное поведение***. Следующий пример иллюстрирует реализацию классического полиморфизма:

```
using System;
using System.Text;
namespace ConsoleApplication7
{

 class Base
 {
     public virtual void F()
     {
       Console.WriteLine("F_Base()");
     }
 }

 class FromBase : Base
 {
     public override void F()
     {
       Console.WriteLine("F_FromBase()");
     }
 }

 class App
 {
     public static void Main()
     {
       Console.Title = "override";
```

```
            Console.BackgroundColor = ConsoleColor.White;
            Console.ForegroundColor = ConsoleColor.Black;
            Console.Clear();

            Base       b =new Base();
            FromBase   fb=new FromBase();

            b.F();
            fb.F();

            Base       b2=fb;
            b2.F();
        }
    }
}
```

Результат выполнения представлен на рис.6.4.

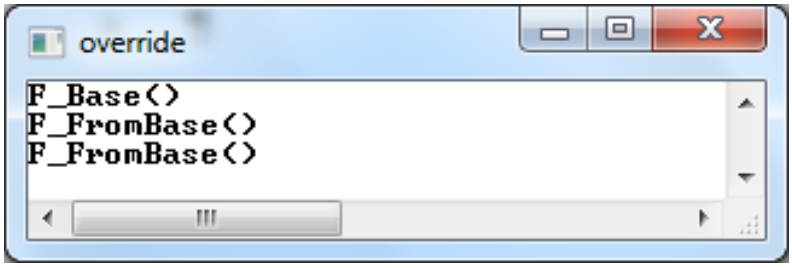

Рис.6.4. Классический полиморфизм

Как видно, при классическом полиморфизме одноименная функция базового класса не доступна в производном классе.

7. АБСТРАКТНЫЙ КЛАСС

В языке C# наряду с обычными классами используются абстрактные классы. Такие классы используются как базовые классы при создании иерархии классов. Своим составом членов и виртуальных функций абстрактные классы определяют функциональную организацию производных классов иерархии. При этом непосредственное создание объектов абстрактных классов бессмысленно и поэтому невозможно. Абстрактные классы специфицируются ключевым словом ***abstract***. Так, следующий пример приведет к ошибке (рис.7.1).

```
using System;

namespace ConsoleApplication1
{

    abstract class Class1
    {
        public void F()
        {
            Console.WriteLine("F_Class1()");
        }
    }

    class App
    {
        public static void Main()
        {
            Class1 c1 = new Class1();

            c1.F();

            Console.Read();
        }
    }
}
```

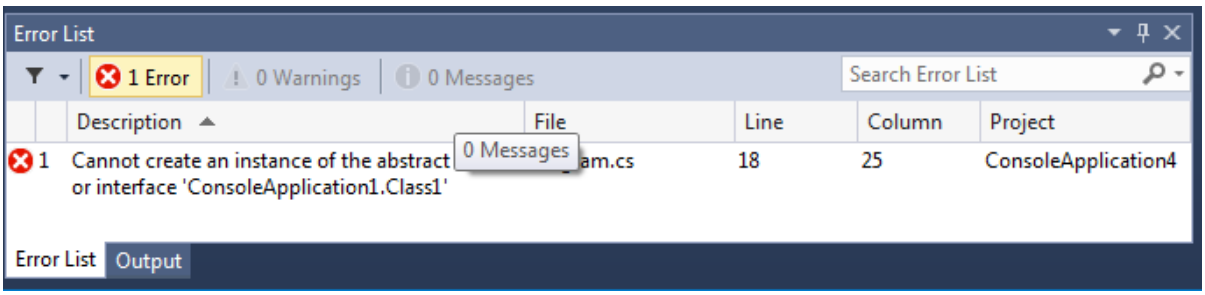

Рис.7.1. Ошибка при создании объекта абстрактнорго класса

Если класс объявлен как абстрактный и мы хотим чтобы при создании производных классов все виртуальные методы базового класса были переопределены, то такие методы можно описывать как абстрактные методы. Абстрактные методы – это те же виртуальные методы, но не требующие в базовом классе своего определения:

```
using System;
namespace ConsoleApplication8
{

    abstract class Abstract
    {
        public abstract void F();

    }

    class Class1 : Abstract
    {
        public void A()
        {
            Console.WriteLine("A()");
        }
    }

    class App
    {
        public static void Main()
        {
```

```
            Class1 c1 = new Class1();
            c1.A();
        }
    }
}
```

Попытка определить класс *Class1* производный от *Abstract* без определения метода *F( )* (рис.7.2).

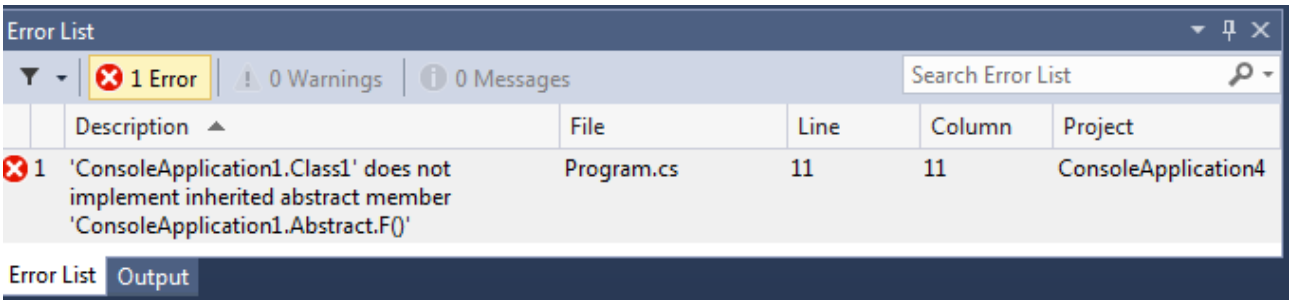

Рис.7.2. Сообщение об ошибке «Необходимо определить абстрактный метод»

Необходимо помнить, что класс, имеющий хотя бы один абстрактный метод, должен быть объявлен как абстрактный.

После исправления ошибок:

```
using System;

namespace ConsoleApplication9
{

    abstract class Abstract
    {
        public abstract void F();

    }

    class Class1 : Abstract
    {
        public void A()
        {
```

```
            Console.WriteLine("A()");
        }
        public override void F()
        {
            Console.WriteLine("F_Class1()");
        }
    }

    class App
    {
        public static void Main()
        {
            Console.Title = "abstract";
            Console.BackgroundColor = ConsoleCol-
or.White;
            Console.ForegroundColor = ConsoleCol-
or.Black;
            Console.Clear();

            Class1 c1 = new Class1();

            c1.A();
            c1.F();

        }
    }
}
```

Получим результат, показанный на рис.7.3

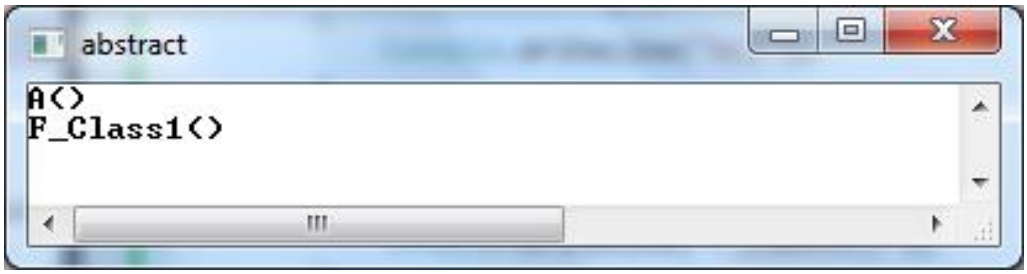

Рис.7.3. Результат после определения абстрактного метода

8. ПРИВЕДЕНИЕ ТИПОВ В ЯЗЫКЕ C#

В C# допустимы операции приведения типов, которые имеют синтаксис:

(тип_приведения) приводимая_переменная

Возможно приведение как числовых типов, так и классов.

При приведении числовых типов в меньшую сторону возможна потеря данных:

```
using System;

namespace ConsoleApplication10
{
    struct s
    {
        public int a;
        public int b;
    }
    class App
    {
        public static void Main()
        {
            Console.Title = "cast";
            Console.BackgroundColor =
                                ConsoleColor.White;
            Console.ForegroundColor =
                                ConsoleColor.Black;
            Console.Clear();

            s s1;
            s1.a = 1;
            s1.b = 2;

            int c = 255 * 255;
```

```
            //c=(int)s1; ошибка

            byte a = (byte)c;

            Console.WriteLine("a={0},c={1}", a, c);
        }
    }
}
```

Результат выполнения представлен на рис.8.1.

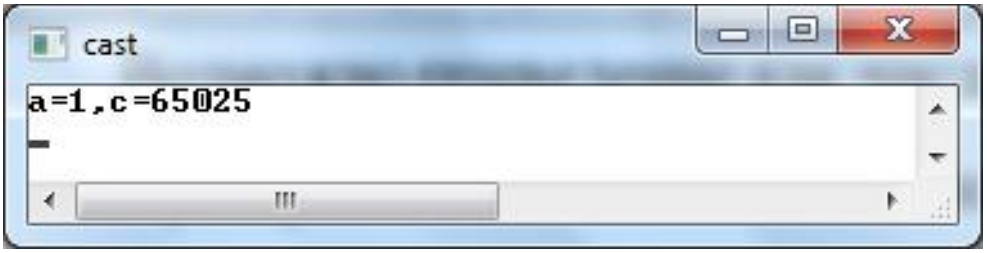

Рис.8.1. Результат приведения типа

Для классов правило приведения типов формулируется так: Если один класс является производным от другого класса, всегда допустимо ссылаться на объект производного класса через ссылку на объект базового класса.

9. КЛАСС КАК МОДУЛЬ

В языке C# разрешается объявить класс, который не рассматривается как тип данных и у которого сохраняется единственная роль – роль модуля. Такой класс объявляется с модификатором ***static***. У такого класса могут быть заданы константы, и только *статические члены*. У него не может быть нестатических конструкторов для создания объектов класса, соответственно он не может рассматриваться как тип данных, для которого можно создавать объекты.

*Замечание***:** Раньше отмечалось, что для статического класса не создаются объекты. Это утверждение не совсем верно, точнее – для статического класса явно не создаются объекты. Один объект (***статический)*** создается автоматически и всегда, не требуя задания операции ***new***, как это делается для динамически создаваемых объектов. Этот объект получает имя, совпадающее с именем класса.

Статический объект, рассматриваемый как модуль класса, создается статическим конструктором, который может быть задан явно, но в случае отсутствия явного задания всегда автоматически добавляется к статическому классу. Этот конструктор вызывается неявно до начала выполнения какой-либо операции над классом. Статический конструктор работает точно так же, как и обычный конструктор. По сути, статический класс – это сервисный класс, предоставляющий свои сервисы пользователям этого класса. Отметим, что классы, объявленные без спецификатора ***static***, но включающие статические и нестатические члены, одновременно могут выступать как ***классом-модулем,*** так и ***классом-типом***. Примером класса модуля в чистом виде служит класс Math.

10. СБОРКА МУСОРА

10.1. Механизм сборки мусора

В .NET Framework применяется схема автоматического управления памятью, которая получила название сборка мусора (garbage collection). Этот механизм позволяет отслеживать неиспользуемые объекты и освобождать занятую ими память совершенно без участия приложения.

Рассмотрим следующий пример:

```
using System;
using System.Text;

namespace GarbageCollection1
{
    class MyClass
    {
        public void Method()
        {
         // какой-то код
        }

    }
    class Program
    {


        public static void GarbageCollectionExample( )
        {
            MyClass   ob = new     MyClass ( );

            ob.Method();

        }

        static void Main(string[] args)
        {

            GarbageCollectionExample1();
```

```
            Console.ReadLine();
        }
    }
}
```

Когда метод *GarbageCollectionExample( )* завершится, переменная *ob* выйдет из области видимости и поскольку она размещается в стеке, то будет уничтожена. Объект, на который ссылалась переменная *ob*, будет уничтожен не сразу, так как он размещается в куче. Поскольку в приложении на объект ссылалась только переменная *ob*, то после завершения метода *GarbageCollectionExample()* объект превращается в мусор, который нужно удалить.

Сборщик мусора непрерывно проверяет дерево ссылок в фоновом режиме и определяет объекты, на которые больше нет ссылок. Обнаружив такой объект (например, *ob* из нашего примера), сборщик мусора удаляет его (используя деструктор класса) и освобождает занятую им память. Поскольку сбор мусора работает постоянно, не обязательно явно уничтожать объекты, как только они станут ненужными.

Обычно сборщик мусора работает в низкоприоритетном потоке, который исполняется, пока процессор не занят более важными задачами, но при дефиците памяти приоритет этого потока повышается. В результате память освобождается быстрее, пока не исчезнет ее дефицит, тогда приоритет потока сборщика мусора вновь снижается.

Подобный подход к освобождению памяти, получивший название недетерминированного, призван обеспечить максимальную производительность приложений и предоставить им окружение, более устойчивое к ошибкам.

Однако за эти удобства приходится расплачиваться неопределенностью. Невозможно с уверенностью сказать, когда именно будет уничтожен тот или иной объект, так как невозможно контролировать момент вызова деструктор класса, подготавливающего объекты к уничтожению. По этой причине в деструкторе не должно быть кода, который необходимо исполнить в заданное конкретное время.

10.2. Управление сборкой мусора

Большинство объектов, используемых в программах на C#, относятся к ***управляемым*** или managed-коду, и легко очищаются сборщиком мусора. Наряду с такими в программах приходится использовать и ***неуправляемые*** объекты (низкоуровневые файловые дескрипторы, сетевые подключения и т.д.), которые обращаются к API операционной системы. Сборщик мусора может справиться с управляемыми объектами, но не знает, как удалять неуправляемые объекты. В этом случае разработчик должен сам реализовывать механизмы очистки в своей программе.

Для освобождения неуправляемых ресурсов имеется два механизма:

1) использование деструктора;
2) использование интерфейса System.IDisposable

и класс System.GC для управления сборкой мусора из программы.

10.2.1 Использование деструктора

Для иллюстрации использования механизма деструктора добавим в класс *MyClass* деструктор:

```
~MyClass()
    {
       Console.Beep();
    }
```

который просто подаёт звуковой сигнал для отслеживания момента его срабатывания, и запустим приложение на исполнение. В реальных программах в деструктор вкладывается логика освобождения неуправляемых ресурсов. Выполнение приложения приостановится на операторе *Console.ReadLine( ),* но дестркутор к этому моменту времени не сработает, о чём будет свидетельствовать отсутствие звукового сигнала. И лишь когда вы осуществите ввод с клавиатуры строки и приложение завершится, только тогда вы услышите сигнал – деструктор выполнился.

Таким образом, сборщик мусора запускается не сразу после удаления всех ссылок на объект, размещенный в куче. Он запускается в то время, когда среда CLR обнаружит в этом потребность, например, когда программе требуется дополнительная память.

10.2.2. Финализируемые объекты

На самом деле при очистке памяти сборщик мусора вызывает не деструктор, а метод *Finalize( )*. Этод метод создаётся компилятором на основе определенного в классе метода деструктора и имеет приводимую ниже структуру:

```
protected override void Finalize( )
{
   try
   {
      // тело  деструктора
   }
   finally
   {
      base.Finalize( );
   }
}
```

Сборщик мусора при размещении объекта в куче определяет, поддерживает ли данный объект метод *Finalize( )*. Если да, ссылка на него сохраняется в специальной таблице, которая называется очередь финализации и используется в дальнейшем для вызова этого метода.

Как правило, объекты в куче располагаются неупорядоченно, между ними могут иметься пустоты. Куча довольно сильно фрагментирована. Поэтому после очистки памяти в результате очередной сборки мусора оставшиеся объекты перемещаются в один непрерывный блок памяти. Вместе с этим происходит обновление ссылок, чтобы они правильно указывали на новые адреса объектов.

Важно также отметить, что для крупных объектов существует своя куча – Large Object Heap. В эту кучу помещаются объекты, размер которых больше 85 Кб. Особенность этой кучи состоит в том, что при сборке мусора сжатие памяти не проводится по причине больших издержек, связанных с размером объектов.

Чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются по поколениям. Всего существует три поколения объектов: 0, 1 и 2.

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора; к поколению 1 – относятся объекты, которые пережили одну сборку, а к поколению 2 – объекты, прошедшие более одной сборки мусора.

Когда сборщик мусора приступает к работе, он сначала анализирует объекты из поколения 0. Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками.

Очевидно, что приложение не сможет продолжать работу, пока не отработает сборщик мусора и на сжатие занятого пространства при сборке мусора требуется время. Несмотря на это, благодаря такому подходу происходит оптимизация приложения. Теперь чтобы найти свободное место в куче среде CLR не надо искать островки пустого пространства среди занятых блоков. Ей достаточно обратиться к указателю кучи, который указывает на свободный участок памяти, что уменьшает количество обращений к памяти. Также важным достоинством сборщика мусора является то, что он успешно справляется с циклическими ссылками, которые раньше были наиболее распространенной причиной утечки памяти.

10.2.3. Класс GC

Для управления сборкой мусора из приложения в библиотеке классов .NET отвечает класс System.GC. Как правило, надобность в применении этого класса отсутствует. Наиболее распространенным случаем его использования является сборка мусора при работе с неуправляемыми ресурсами, при интенсивном выделении больших объемов памяти, при которых необходимо такое же быстрое их освобождение. Рассмотрим некоторые методы и свойства класса System.GC:

AddMemoryPressure( ) информирует среду CLR о выделении большого объема неуправляемой памяти, которую надо учесть при планировании сборки мусора. В связке с этим методом используется метод *RemoveMemoryPressure( ),* который указывает CLR, что ранее выделенная память освобождена и ее не надо учитывать при сборке мусора;

Collect( ) приводит в действие механизм сборки мусора. Перегруженные версии метода позволяют указать поколение объектов, вплоть до которого надо произвести сборку мусора;

GetGeneration(Object) позволяет определить номер поколения, к которому относится переданный в качестве параметра объект;

GetTotalMemory( ) возвращает объем памяти в байтах, который занят в управляемой куче;

WaitForPendingFinalizers( ) приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора.

Перед вызовом сборки мусора через *GC.Collect( )* следует вызывать метод *GC.WaitForPendingFinalizers( ),* позволяющий освобождаемым объектам выполнить код по очистке памяти, который определен в методах и деструкторах данных объектов.

С помощью перегруженных версий метода *Collect( )* класса *GC* можно выполнить более точную настройку сборки мусора. Так, его перегруженная версия принимает в качестве параметра число – номер поколения, вплоть до которого надо выполнить очистку. Например, *GC.Collect(0)* - удаляются только объекты поколения 0.

Еще одна перегруженная версия принимает еще и второй параметр типа перечисления *GCCollectionMode*, который может принимать три значения:

Default – значение по умолчанию (*Forced*);

Forced – вызывает немедленное выполнение сборки мусора;

Optimized – позволяет сборщику мусора определить, является ли текущее время оптимальным для освобождения объектов.

Например, немедленная сборка мусора вплоть до первого поколения объектов вызывается так: *GC.Collect(1, GCCollectionMode.Forced);*

Работа с методами System.GC не вызывает большого труда. Изменим в рассмотренном выше примере метод *Main( ):*

```
    static void Main(string[] args)
    {
      Console.Title = "GarbageCollection";
      Console.BackgroundColor = ConsoleCol-
or.White;
      Console.ForegroundColor = ConsoleCol-
or.Black;
      Console.Clear();

      GarbageCollectionExample();

      long totalMemory =
                    GC.GetTotalMemory(false);

      Console.WriteLine(
      "totalMemory={0}\nЗапускаем сборку мусора
      totalMemory);

      GC.WaitForPendingFinalizers();
      GC.Collect();

      Console.ReadLine();
    }
```

После запуска на исполнение (рис.10.1) вы услышите звуковой сигнал до момента начала выполнения оператора *Console.ReadLine( )*, а значит, мы можем отчасти управлять процессом сборки мусора.

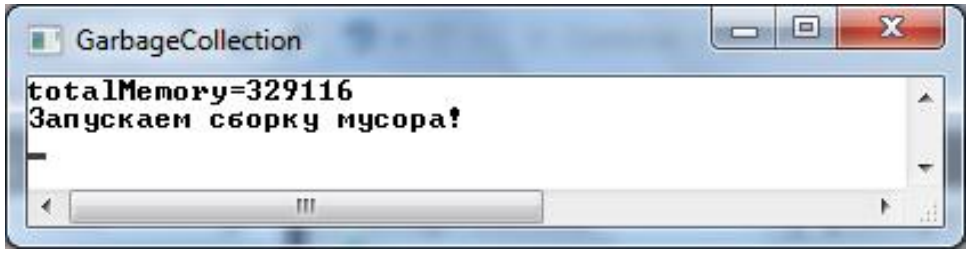

Рис.10.1. Иллюстрация запуска сборки мусора

10.2.4. Интерфейс IDisposable

Конечно, можно принудительно из программы запускать механизм сборки мусора, если необходимо немедленно уничтожить объект и освободить связанные с объектом неуправляемые ресурсы. Но процедура сборки мусора является достаточно затратной.

Лучшим решением проблемы немедленного освобождения неуправляемых ресурсов является определение и использование интерфейса *IDisposable*, предусматривающего один-единственный метод *Dispose( )*, в котором и должно происходить освобождение неуправляемых ресурсов.

Ниже приводится пример с использованием интерфейса *IDisposable:*

```
using System;
using System.Text;

namespace GarbageCollection2
{
  class MyClass : IDisposable
  {
    public void Method()
    {
    }

    public void Dispose()
```

```
    {
      // действия освобождения неуправляемых ресурсов
      Console.Beep();
    }
  }
  class Program
  {
    public static void    GarbageCollectionExample( )
    {
      MyClass     ob=null;

      try
      {
        ob = new MyClass();

        ob.Method();
      }
      finally
      {
        if (ob != null)
          ob.Dispose();
      }
    }

    static void Main(string[] args)
    {
      Console.Title = " IDisposable";
      Console.BackgroundColor = ConsoleColor.White;
      Console.ForegroundColor = ConsoleColor.Black;
      Console.Clear();

      GarbageCollectionExample1();

      Console.ReadLine();
    }
  }
}
```

Отметим, что в приведенной реализации класса *MyClass* мы вовсе отказались от деструктора.

Запустив программу на исполнение, можно убедиться, что:

1) метод *Dispose( )* освобождает неуправляемые ресурсы (звучит звуковой сигнал);
2) метод выполняется до вызова оператора *Console.ReadLine( ).*

Таким образом, в отличие от использования деструктора метод *Dispose( )* позволяет освобождать критические ресурсы в любой момент времени и менее дорогим способом, чем принудительная сборка мусора.

При определении метода *GarbageCollectionExamplel( )* была использована конструкция *try...finally*, которая по сути эквивалентна следующим двум строкам кода:

```
MyClass ob = new MyClass();
ob.Dispose();
```

Но конструкцию *try...finally* предпочтительнее использовать, так как она гарантирует, что освобождение ресурсов в методе *Dispose( )* произойдет даже в случае возникновения исключения.

Альтернативой конструкции *try...finally* может служить конструкция:

```
using (MyClass   ob= new MyClass())
   {
       ob.Method();
   }
```

при завершении выполнения тела которой автоматический вызывается метод *Dispose( )* для объекта *ob*.

10.3. Комбинирование подходов

Из двух рассмотренных механизмов использование интерфейса *IDisposable* кажется предпочтительным для освобождения неуправляемых

ресурсов, поскольку метод *Dispose( )* можно вызвать в любой момент, а значит, в отличие от деструктора освободить критические ресурсы, когда потребуется. Но что случится, если программист вдруг забудет поставить вызов метода *Dispose( )?* Как устранение этой негативной возможности рекомендуется использовать следующий формализованный шаблон от компании Microsoft, реализующий кобинированный подход как сочетание использования деструктора и метода *Dispose( ):*

```
public class Base: IDisposable
{
   private bool disposed = false;
   // реализация интерфейса IDisposable.
   public void Dispose()
   {
      Dispose(true);
      // подавляем финализацию
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing)
   {
      if (!disposed)
      {
         if (disposing)
         {
            // Освобождаем управляемые ресурсы
         }
         // освобождаем неуправляемые объекты
         disposed = true;
      }
   }
   // Деструктор
   ~Base()
   {
      Dispose (false);
   }
}
```

Шаблон преполагает наличие перегруженной версии метода *Dispose(bool disposing)*. При вызове деструктора в качестве параметра *disposing* передается значение *false*, чтобы избежать очистки управляемых ресурсов, так как мы не можем быть уверенными в их состоянии, а за их освобождение отвечает сборщик мусора.

Использование в методе *Dispose( )* метода *GC.SuppressFinalize(this)* блокирует вызов метода *Finalize( )* для данного объекта.

Таким образом, даже если разработчик не использует в программе метод *Dispose( )*, все равно произойдет очистка и освобождение ресурсов.

Для использования комбинированного подхода существуют следующие общие рекомендации:

1) деструктор следует реализовывать только для тех объектов, которым он действительно необходим, так как метод *Finalize( )* оказывает сильное влияние на производительность;

2) после вызова метода *Dispose( )* необходимо блокировать у объекта вызов метода *Finalize( )* с помощью *GC.SuppressFinalize( );*

3) при создании производных классов от базовых, которые реализуют интерфейс *IDisposable*, следует также вызывать метод *Dispose( )* базового класса:

```
public class Derived: Base
{
   private bool IsDisposed = false;
   protected override void Dispose(bool disposing)
   {
      if (!IsDisposed)
      {
         if (disposing)
         {
            // Освобождение управляемых ресурсов
```

```
            }
            IsDisposed = true;
        }
        // Обращение к методу Dispose базового класса
        base.Dispose(disposing);
    }
}
```

4) отдавайте предпочтение комбинированному шаблону, реализующему как метод *Dispose( )*, так и деструктор.

5)

10.4. Пример

Ниже приводится полный пример использования комбинированного подхода для классов, связанных наследованием (результат показан на рис.10.2):

```
using System;
using System.Text;


namespace GarbageCollection3
{

  class Base : IDisposable
  {
    public void BaseMethod()
    {
      Console.WriteLine("BaseMethod() working... ");
    }

    private bool disposed = false;

    // реализация интерфейса IDisposable.
```

```
public void Dispose()
{
  Dispose(true);

  // подавляем финализацию
  GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
  if (!disposed)
  {
    if (disposing)
    {
      // Освобождаем управляемые ресурсы
      Console.WriteLine(
        "Base: Освобождаем управляемые ресурсы ");
      Console.Beep();
    }
    // освобождаем неуправляемые объекты
    disposed = true;
  }
}

// Деструктор
~Base()
 {
   Dispose(false);
 }
}

class Derived : Base
{

  private bool IsDisposed = false;

  protected override void Dispose(bool disposing)
 {

   if (!IsDisposed)
```

```
    {
      if (disposing)
      {
        // Освобождаем управляемые ресурсы
        Console.WriteLine("Derived: Освобождаем
                           управляемые ресурсы ");
        Console.Beep();
      }
      IsDisposed = true;
    }

    // Обращение к методу Dispose базового класса
    base.Dispose(disposing);
  }

  public void DerivedMethod()
  {
    BaseMethod();
    Console.WriteLine("DerivedMethod()
                                  working...");
  }

}

class Program
{
  public static void  GarbageCollectionExamplel( )
  {
    using (Derived    ob= new Derived())
    {
      ob.DerivedMethod();
    }

  }

  static void Main(string[] args)
  {

    Console.Title = "Combined";
    Console.BackgroundColor = ConsoleColor.White;
    Console.ForegroundColor = ConsoleColor.Black;
```

```
            Console.Clear();

            Console.WriteLine("Main() start... ");

            GarbageCollectionExamplel();

            Console.WriteLine("Input: ");
            Console.ReadLine();

            Console.WriteLine("Main() end... ");
        }
    }
}
```

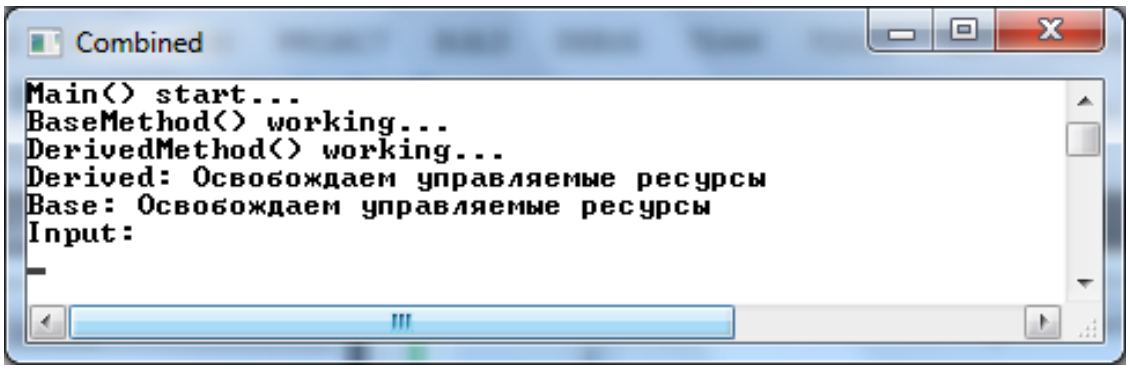

Рис.10.2. Иллюстрация комбинированного подхода

11. ВОПРОСЫ ДЛЯ КОНТРОЛЯ

1. Отличие класса типа от класса модуля.
2. Синтаксис объявления класса.
3. Сколько родителей может иметь класс в языке C #?
4. Какие члены может иметь класс в языке C #?
5. Синтаксис и назначение конструктора и деструктора класса.
6. Принципы объектно-ориентированного программирования в языке C #.
7. Как вы понимаете термин инкапсуляция?
8. Как вы понимаете термин наследование?
9. Как вы понимаете термин полиморфизм?
10. Какие данные-члены класса называют полями?
11. Статические члены класса и обращение к ним.
12. Объектные методы класса и обращение к ним.
13. Поля, ограничение доступа к полям класса.
14. Как реализуется инкапсуляция данных в классах на языке C#?
15. Свойства класса.
16. Типы наследования в языке C#.
17. Какое назначение this и base?
18. Как запретить наследование от класса?
19. Какое назначение вложенных классов?
20. Как реализуется замещение функции базового класса на языке C#?
21. Как реализуется полиморфизм на языке C#?
22. Какое назначение абстрактных классов?
23. Как реализуется операция приведения типов для ссылочных типов?
24. Назначение и принцип действия механизма сборки мусора?
25. Как можно управлять удалением объектов в среде .Net?
26. Назовите механизмы освобождения неуправляемых ресурсов в среде .Net?
27. Каковое назначение метода *Finalize( )?*
28. Назначение и возможности класса System.GC?
29. Состав и назначение интерфейса IDisposable?
30. В чём заключается комбинированный подход для освобождения неуправляемых ресурсов?

ПРИЛОЖЕНИЕ

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Для выполнения лабораторных работ могут быть использованы системы программирования MS Visual Studio .Net 2008, 2010, 2012, 2013, 2015 MS Visual C# Express Edition 2005, 2008, 2010, 2012, 2013.

По результатам выполнения каждой лабораторной работы студент должен подготовить и представить преподавателю отчет, который включает разделы:

– постановка задачи;

– метод решения задачи;

– описание разработанных классов (структура данных и алгоритмы);

– описание программы;

– тестовый пример;

– результаты, выдаваемые программой;

– выводы.

ЛАБОРАТОРНАЯ РАБОТА 1

Разработка класса как типа данных

Цель работы – освоение принципов объектно-ориентированного программирования в среде .NET Framework на языке C# на примере разработки класса как типа.

Задание

1) Изучить принципы объектно-ориентированного программирования в среде .NET Framework на языке C#.

2) Разработать и программно реализовать классы для решения предложенной задачи (варианты в табл.П1).

3) Использовать разработанный класс для решения поставленной задачи.

4) Получить навыки объектно-ориентированного программирования в среде .NET Framework на языке C# для разработки класса как типа.

Порядок выполнения работы

1. Пункт 1 задания изучить по соответствующим разделам данных методических указаний.

2. Пункты 2– 3 задания отработать на примере решения задачи из списка задач.

Требования к программной реализации

Программная реализация должна включать два класса Program и разработанный класс. Класс Program – содержит функцию Main( ) и организует работу приложения:

- вводит исходные данные задачи;
- использует разработанный класс для решения задачи;
- выводит результаты на консоль.

Разработанный класс должен содержать:

- поля с(без) ограничения доступа, переменные и константы;
- закрытые данные-члены и методы доступа к ним;
- свойства, доступные на чтение, запись или чтение и запись;
- объектные и статические методы, реализующие необходимую функциональность класса;
- конструкторы.

Таблица П1 – Варианты задания

№ п/п	Задача
1	Класс для описания фигуры «линия» на координатной плоскости
2	Класс для описания фигуры «окружность» на координатной плоскости
3	Класс для описания фигуры «квадрат» на координатной плоскости
4	Класс для описания фигуры «параллелограм» на координатной плоскости
5	Класс для описания фигуры «эллипс» на координатной плоскости
6	Класс для описания фигуры «ломаная линия» на координатной плоскости
7	Класс для описания фигуры «ромб» на координатной плоскости
8	Класс для описания фигуры «линия» в трехмерном пространстве
9	Класс для описания фигуры «ломаная линия» в трехмерном пространстве
10	Класс для описания фигуры «сфера» в трехмерном пространстве

Продолжение табл.П1

№ п/п	Задача
11	Класс для описания фигуры «куб» в трехмерном пространстве
12	Класс для описания фигуры «параллелепипед» в трехмерном пространстве
13	Класс для операций над комплексными числами
14	Класс для операций над числами в виде дроби (числитель/знаменатель)
15	Класс множество объектов
16	Класс полином
17	Класс матрица
18	Класс функция
19	Класс «длинное целое» число
20	Класс односвязный список
21	Класс двусвязный список
22	Класс стек
23	Класс очередь
24	Класс список с ключами
25	Класс словарь

Для классов, реализующих тип «плоская» фигура реализовать методы:

- расчёта периметра фигуры;
- площади фигуры;
- координат вершин фигуры.

Для классов, реализующих тип «объемная» фигура реализовать методы:

- расчёта площади поверхности фигуры;
- расчёта объема фигуры.

ЛАБОРАТОРНАЯ РАБОТА 2

Разработка класса как модуля.

Цель работы – освоение принципов объектно ориентированного программирования в среде .NET Framework на языке C# на примере разработки класса как модуля.

Задание

1) Изучить принципы объектно-ориентированного программирования в среде .NET Framework на языке C#.

2) Разработать и программно реализовать класс для решения предложенной задачи (варианты в табл.П2).

3) Использовать разработанный класс для решения поставленной задачи.

4) Получить навыки объектно-ориентированного программирования в среде .NET Framework на языке C# для разработки класса как модуля.

Таблица П2 – Варианты задания

№ п/п	Задача для класса GraphTask	Форма представления графа в классе Graph
1	2	3
1	Поиск пути из вершины в вершину	Список дуг
2	Поиск минимального пути из вершины в вершину	Матрицей смежности
3	Поиск цикла, проходящего через вершину	Список последователей
4	Поиск гамильтонова цикла	Список предшественников
5	Поиск всех циклов, проходящих через вершину	Список последователей и список предшественников
6	Расчет сетевого графика	Список дуг
7	Проверка графа на ациклич-ность	Матрицей смежности
8	Ранжирование ацикличного графа	Список последователей
9	Раскраска графа	Список предшественников
10	Выделение остовного дерева	Список последователей и список предшественников
11	Поиск пути из вершины в вершину	Список дуг
12	Поиск минимального пути из вершины в вершину	Матрица смежности
13	Поиск цикла, проходящего через вершину	Список последователей
14	Поиск гамильтонова цикла	Список предшественников

Продолжение табл.П2

1	2	3
15	Поиск всех циклов, проходящих через вершину	Список последователей и список предшественников
16	Расчет сетевого графика	Список дуг
17	Проверка графа на ацикличность	Матрица смежности
18	Ранжирование ацикличного графа	Список последователей
19	Раскраска графа	Список предшественников
20	Выделение остовного дерева	Список последователей и список предшественников
21	Поиск пути из вершины в вершину	Список дуг
22	Поиск минимального пути из вершины в вершину	Матрица смежности
23	Поиск цикла, проходящего через вершину	Список последователей
24	Поиск гамильтонова цикла	Список предшественников
25	Поиск всех циклов, проходящих через вершину	Список последователей и список предшественников

Порядок выполнения работы

1. Пункт 1 задания изучить по соответствующим разделам данных методических указаний.

2. Пункты 2-3 задания отработать на примере решения задачи из списка задач.

Требования к программной реализации

Программная реализация должна включать три класса Program, класс *Graph* и класс *GraphTask*. Класс *Program* – содержит функцию *Main( )* и организует работу приложения:

- вводит исходные данные задачи;
- использует разработанный класс для решения задачи;
- выводит результаты на консоль.

Класс-тип *Graph* представляет описание объекта граф.

Класс-модуль *GraphTask* обеспечивает функциональность для решения различных задач на графе.

ЛАБОРАТОРНАЯ РАБОТА 3

Разработка иерархии классов.

Цель работы – освоение принципов объектно-ориентированного программирования в среде .NET Framework на языке C# на примере разработки иерархии классов.

Задание

1) Изучить принципы объектно-ориентированного программирования в среде .NET Framework на языке C#.

2) Разработать и программно реализовать классы для решения предложенной задачи.

3) Использовать разработанный класс для решения поставленной задачи.

4) Получить навыки объектно-ориентированного программирования в среде .NET Framework на языке C# для разработки иерархии классов.

Порядок выполнения работы

1. Пункт 1 задания изучить по соответствующим разделам данных методических указаний.

2. Пункты 2–3 задания отработать на примере решения задачи.

Требования к программной реализации

Программная реализация должна включать два класса Program и разработанную иерархию классов для решения предложенной задачи. Класс Program – содержит функцию Main( ) и организует работу приложения:

– вводит исходные данные задачи;

– использует разработанный класс для решения задачи;

– выводит результаты на консоль.

Задача.

Необходимо разработать иерархию классов для графического редактора. Графический редактор работает с изображением как с композицией плоских и объемных фигур. Изображение в памяти представляется как список объектов фигур и может быть сохранено в файле рисунка. Для такого редактора необходимо разработать иерархию классов графических фигур: точка, линия, ломаная линия, окружность, квадрат, параллелограм, эллипс, куб, параллелепипед, сфера.

Для классов, реализующих тип «плоская» фигура реализовать полиморфные методы:

- расчёта периметра фигуры;
- площади фигуры;
- координат вершин фигуры.

Для классов, реализующих тип «объемная» фигура, реализовать полиморфные методы:

- расчёта площади поверхности фигуры;
- расчёта объема фигуры.

ЛАБОРАТОРНАЯ РАБОТА 4

Управление сборкой мусора

Цель работы – изучение и освоение возможностей управления сборкой мусора в программах на языке C# в среде .NET.

Задание

1) Изучить принципы работы механизма сборки мусора в программах на языке C# в среде .NET Framework.

2) Изучить механизмы освобождения неуправляемых ресурсов.

3) Изучить возможности класса *System.GC*.

4) Изучить назначение и использование интерфейса *IDisposeable*.

5) Освоить суть и принципы использования комбинированного подхода для удаления объектов.

6) Отработать полученные знания при решении задачи.

Порядок выполнения работы

1. Пункты задания 1–5 изучить по соответствующим разделам данных методических указаний.

2. Пункт 6 задания отработать на примере решения задачи.

Задача.

Использовать разработанную в лабораторной работе 3 иерархию классов для разработки графического редактора. Разработать классы *Picture* и *GRedaktor*.

Требования к программной реализации

Для класса *Picture* реализовать методы:

- сохранения изображения из памяти в файле рисунка;
- загрузки изображения из файла в память;
- реализовать комбинированный подход удаления объектов.

СПИСОК ЛИТЕРАТУРЫ

1. Рейли Д. Создание приложений Microsoft ASP.Net / Д.Рейли : пер.с англ.– М.: Изд.-торговый дом «Русская редакция», 2002.– 480с., ил.

2. Петцольд Ч. Программирование для Microsoft Windows на C# : В 2-х т. Т.1. / Ч. Петцольд : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 576 с., ил.

3. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.2. / Ч. Петцольд» : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002.– 624 с., ил.

4. Лабор В. В. Си Шарп : Создание приложений для Windows / В. В. Лабор.– Мн.: Харвест, 2003.– 384 с.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework / Дж. Рихтер : пер. с англ. – 2-е изд., испр. – М.: Издательско-торговый дом «Русская Редакция», 2003.– 512 с., ил.

6. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# -NET : учеб. курс MCAD/MCSD : пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2003.– 512 с., ил.

7. Троелсен Э. C# и платформа NET. Библиотека программиста / Э. Троелсен. – СПб.: Питер, 2003. – 800 с.,ил.

8. Анализ требований и создание архитектуры решений на основе Microsoft .NET : учеб. курс MCSD/пер. с англ.–М.: Издательско-торговый дом «Русская Редакция», 2004.– 416 с., ил.

9. Шилдт Г. Полный справочник по C# / Шилдт Г. : пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 752 с., ил.

10. Бишоп Дж. C# в кратком изложении / Дж.Бишоп, Н.Хорспул : пер. с англ.– М.: Бином, Лаборатория знаний, 2005. – 472 с., ил.

СОДЕРЖАНИЕ

Навчальне видання

Методичні вказівки

до лабораторних робіт

«Об’єктно-орієнтоване програмування мовою C#»

з дисципліни «Технології програмування»

для студентів спеціальностей
122 – Комп’ютерні науки та інформаційні технології,
124 – Системний аналіз, в тому числі для іноземних студентів

Російською мовою

Укладачі:

КОЖИН Юрій Миколайович

МАЛИХ Олег Миколайович

ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск О. С. Куценко

Роботу до друку рекомендував О. В. Горілий

Редактор О. І. Шпільова

План 2017 , поз. 1

Підп. до друку Формат 60х84 1/16 Папір офсетний.

Riso-друк. Гарнітура Таймс. Ум.друк.арк. 2,8

Наклад 100 прим. Зам. № Ціна договірна.

Видавничий центр НТУ “ХПІ”,

61002, м.Харків, вул. Кирпичова, 2

Свідоцтво суб’єкта про реєстрацію ДК №3657 від 24.12.2009 р.

Друкарня НТУ “ХПІ” . 61002, м.Харків, вул. Кирпичова, 2