Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ, ПРОГРАММИРОВАНИЕ

сборник учебно-методических материалов
для направления подготовки 38.03.05 Бизнес-информатика

Благовещенск

2017

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Составитель: Галаган Т.А.

Объектно-ориентированный анализ, программирование: сборник учебно-методических материалов для направления подготовки 38.03.05 Бизнес-информатика – Благовещенск: Амурский гос. ун-т, 2017. – 74 с.

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

Структура программы на языке C# и метод Main( )

Программа начинается с директивы *using System*;

Она позволяет использовать имена стандартных классов из пространства *System* непосредственно, не требуя явного указания имени пространства.

Использование ключевого слова *namespace* необходимо при создании для проекта собственного пространства имен, названного по умолчанию *ConsoleApplication1*. Это обеспечивает возможность давать элементам программы имена, не отслеживая их совпадение с именами из других пространств имен.

Любая программа на языке C# состоит из описания классов и их взаимодействия. Если программа состоит из одного класса, ему по умолчанию задается имя *Class1*.

После запуска программы управление передается методу, который носит имя *Main*. Он обязательно должен быть статическим. Он также может принимать параметры и возвращать значение. Если *Main* не возвращает значение, в качестве его типа указывается *void*. Если *Main* имеет параметры, они передаются в массив *args*.

Пример простейшей программы:

```
using System;
namespace ConsoleApplication1
{
   class Class1
   {
      static void Main( )
      {
         Console.WriteLine("Первая программа"); // вывод сообщения на экран
      }
   }
}
```

Объявление классов и создание объектов

Раньше программисты, в большинстве случаев, использовали функциональный или процедурный принцип программирования. Все программы, большие и маленькие, писались в одном файле. С течением времени программы становились всё сложнее и больше, что доставляло проблемы разработчикам при поддержке таких программ и внесении изменений. Эту проблему решает объектно-ориентированное программирование. ООП позволяет объединить данные и методы, относящиеся к одной сущности, и работать с ними, как с одним целым.

Классы и объекты

ООП привносит нам два ключевых понятия: **Класс** и **Объект**. **Класс** – это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д. Описав класс, мы можем создать его экземпляр – **объект**. Объект – это уже конкретный представитель класса.

Пример. Допустим, нам в программе необходимо работать со странами. Страна – это абстрактное понятие. У нее есть такие характеристики, как название, население, площадь, флаг и другое. Для описания такой страны будет использоваться класс с соответствующими полями данных. Такие страны, как Россия и Украина будут уже объектами (конкретными представителями типа страна).

Общие положения объектного подхода

Анализ предметной области необходим до начала написания программы. Объектный подход к разработке сложных программных систем предполагает необходимость этапов анализа и моделирования предметной области до непосредственного кодирования программы на каком-либо языке программирования.

Объектный подход применяется на всех основных стадиях жизненного цикла программного обеспечения и включает в себя несколько этапов.

Объектно-ориентированный анализ – методология анализа предметной области, который выполняется с целью выделения объектов и классов в качестве требования к проектируемой системе.

Объектно-ориентированное проектирование – методология проектирования, объединяющая в себе процесс декомпозиции объектов и приемы их представления логической и физической моделью проектируемой системы.

Объектно-ориентированное программирование – методология (парадигма программирования), в основе которой лежит построение программы в форме взаимодействия объектов, каждый из которых является экземпляром определенного класса.

Под объектом понимают некоторую сущность, обладающую определённым состоянием и поведением, имеющую заданные значения свойств (атрибутов) и операций над ними (методов).

Объект определяется через его внешнее отличие от других объектов. Внутренняя особенность объекта (его структура, внутренние характеристики) не влияет на внешнее отличие и для объектного моделирования значения не имеет.

Главная цель объектного анализа – представление предметной области в виде множества объектов со своими свойствами и характеристиками, которые достаточны для их определения и идентификации, а также для задания поведения объектов в рамках выбранной системы понятий и абстракций. Каждый объект – это уникальный элемент, имеющий, по крайней мере, одно свойство или характеристику и уникальную идентификацию во множестве объектов.

Предметная область может являться самостоятельным объектом или быть объектом в составе другой предметной области.

Класс представляет собой множество объектов, обладающих одинаковыми свойствами и операциями, отношениями и семантикой. Любой объект является экземпляром класса.

Принципы декомпозиции и абстрагирования

Принцип декомпозиции. Декомпозиция – это разбиение целого на составные элементы. В рамках объектного подхода рассматривают два вида декомпозиции: алгоритмическую и объектную.

В соответствии с алгоритмической декомпозицией предметной области при анализе задачи разработчик пытается понять, какие алгоритмы необходимо разработать для ее решения, каковы спецификации этих алгоритмов (вход, выход), и как эти алгоритмы связаны друг с другом. В языках программирования данный подход в полной мере поддерживается средствами модульного программирования (библиотеки, модули, подпрограммы).

Объектная декомпозиция предполагает выделение основных содержательных элементов задачи, разбиение их на типы (классы), определение свойств (данные) и поведения (операции) для каждого класса его, а также взаимодействия классов друг с другом. Объектная декомпозиция поддерживаются всеми современными объектно-ориентированными языками программирования.

Принцип абстрагирования. Абстрагирование применяется при решении многих задач - любая модель позволяет абстрагироваться от реального объекта, подменяя его изучение исследованием формальной модели. Абстрагирование в ООП позволяет выделить основные элементы предметной области, обладающие одинаковой структурой и поведением. Такое разбиение предметной области на абстрактные классы позволяет существенно облегчить анализ и проектирование системы.

Согласно этому принципу в модель включаются только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций.

Основные принципы объектно-ориентированного программирования

ООП основывается на нескольких базовых принципах, каждому из которых будет посвящен отдельный урок, а пока коротко рассмотрим их.

Инкапсуляция – позволяет скрывать внутреннюю реализацию. В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым доступ для пользователя необходимо запретить, тут и используется инкапсуляция.

Наследования – позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником. Например, есть базовый класс животное. В нем описаны общие характеристики для всех животных (класс животного, вес). На базе этого класса можно создать классы наследники Собака, Слон со своими специфическими свойствами. Все свойства и методы базового класса при наследовании переходят в класс наследник.

Полиморфизм – это способность объектов с одним интерфейсом иметь различную реализацию. Например, есть два класса, Круг и Квадрат. У обоих классов есть метод *GetSquare()*, который считает и возвращает площадь. Но площадь круга и квадрата вычисляется по-разному, соответственно, реализация одного и того же метода различная.

Абстракция – позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект. Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т.д.

Описание класса

Класс – это абстрактный тип данных. Другими словами, класс – это некоторый шаблон, на основе которого будут создаваться его экземпляры – **объекты**.

В C# классы объявляются с помощью ключевого слова *class*. Общая структура объявления выглядит следующим образом:

```
[модификатор доступа] class [имя_класса]
{
   //тело класса
}
```

Класс следует объявлять внутри пространства имен *namespace*, но за пределами другого класса. Пример объявления классов *Student* и *Pupil*:

```
namespace HelloWorld
{
   class Student //без указания модификатор доступа, класс будет internal
   {
      //тело класса
   }
   public class Pupil
   {
      //тело класса
   }
}
```

Классы в C# могут содержать следующие элементы:

- поля;
- константы;

- свойства;
- конструкторы;
- методы;
- события;
- операторы;
- индексаторы;
- вложенные типы.

Все элементы класса, как и сам класс, имеют свой уровень доступа.

–*public* – доступ к члену возможен из любого места одной сборки, либо из другой сборки, на которую есть ссылка;

–*protected* – доступ к члену возможен только внутри класса, либо в классе-наследнике (при наследовании);

–*internal* – доступ к члену возможен только из сборки, в которой он объявлен;

–*private* – доступ к члену возможен только внутри класса;

–*protected internal* - доступ к члену возможен из одной сборки, либо из класса-наследника другой сборки.

При объявлении класса модификатор доступа можно не указывать, при этом будет применяться режим по умолчанию *internal*.

Не указав модификатор доступа для члена, по умолчанию ему будет присвоен режим *private*.

При помощи модификаторов доступа в C# реализуется один из базовых принципов ООП – **инкапсуляция**.

Поля класса

Поле – это переменная, объявленная внутри класса. Как правило, поля объявляются с модификаторами доступа *private* либо *protected*, чтобы запретить прямой доступ к ним. Для получения доступа к полям следует использовать свойства или методы.

Пример объявления полей в классе:

```
class Student
{
   private string firstName;
   private string lastName;
   private int age;
   public string group; // не рекомендуется использовать public для поля
}
```

Константы

Константа – это переменная, значений которой нельзя изменить. Константа объявляется с помощью ключевого слова *const*. Пример объявления константы:

```
class Math
{
   private const double Pi = 3.14;
}
```

Создание объектов

Объявив класс, можно создавать объекты. Делается это при помощи ключевого слова *new* и имени класса:

Доступ к членам объекта осуществляется при помощи оператора точка «**.**»:

```
static void Main(string[] args)
{
   Student student1 = new Student();
```

```
    Student student2 = new Student();

    student1.group = "Group1";
    student2.group = "Group2";

    Console.WriteLine(student1.group); // выводит на экран "Group1"
    Console.Write(student2.group);
    Console.ReadKey();
}
```

Такие поля класса *Student*, как *firstName*, *lastName* и *age* указаны с модификатором доступа *private*, поэтому доступ к ним будет запрещен вне класса:

```
static void Main(string[] args)
{
    Student student1 = new Student();
    student1.firstName= "Nikolay"; //ошибка, нет доступа к полю firstName.
                                   //Программа не скомпилируется
}
```

Конструкторы

Конструктор – это метод класса, предназначенный для инициализации объекта при его создании.

Инициализация – это задание начальных параметров объектов/переменных при их создании.

Особенностью конструктора, как метода, является то, что его имя всегда совпадает с именем класса, в котором он объявляется. При этом, при объявлении конструктора, не нужно указывать возвращаемый тип, даже ключевое слово *void*. Конструктор следует объявлять как *public*, иначе объект нельзя будет создать (хотя иногда в этом также есть смысл).

В классе, в котором не объявлен ни один конструктор, существует неявный конструктор по умолчанию, который вызывается при создании объекта с помощью оператора *new*.

Объявление конструктора имеет следующую структуру:

```
public [имя_класса] ([аргументы])
{
    // тело конструктора
}
```

Например, есть класс *Автомобиль*. Создавая новый автомобиль, значения пробега и количества топлива в баке есть смысл поставить равными нулю:

```
class Car
{
    private double mileage;
    private double fuel;
    public Car() //объявление конструктора
    {
        mileage = 0;
        fuel = 0;
    }
}
class Program
{
    static void Main(string[] args)
    {
```

```
        Car newCar = new Car(); // создание объекта и вызов конструктора
    }
}
```

Без конструктора пришлось бы после создания объекта отдельно присваивать значения его полям, что очень неудобно.

Конструктор также может иметь параметры.

Пример с тем же автомобилем, только теперь при создании объекта мы можем задать любые начальные значения:

```
class Car
{
    private double mileage;
    private double fuel;

    public Car(double mileage, double fuel)
    {
        this.mileage = mileage;
        this.fuel = fuel;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car newCar = new Car(100, 50); //вызов конструктора с параметрами
    }
}
```

Ключевое слово this

В примере выше используется ключевое слово *this*.

Указатель *this* - это указатель на объект, для которого был вызван нестатический метод. Ключевое слово *this* обеспечивает доступ к текущему экземпляру класса. Классический пример использования *this*, это как раз в конструкторах, при одинаковых именах полей класса и аргументов конструктора. Ключевое слово *this* это что-то вроде имени объекта, через которое мы имеем доступ к текущему объекту.

Несколько конструкторов

В классе возможно указывать множество конструкторов, главное чтобы они отличались сигнатурами. Сигнатура, в случае конструкторов, - это набор аргументов. Например, нельзя создать два конструктора, которые принимают два аргумента типа *int*.

Пример использования нескольких конструкторов:

```
class Car
{
   private double mileage;
   private double fuel;

   public Car()
   {
     mileage = 0;
     fuel = 0;
   }

  public Car(double mileage, double fuel)
```

```
  {
    this.mileage = mileage;
    this.fuel = fuel;
  }
}
class Program
{
  static void Main(string[] args)
  {
    Car newCar = new Car(); // создаем автомобиль с
                            //параметрами по умолчанию, 0 и 0
    Car newCar2 = new Car(100, 50); // создаем автомобиль с указанными параметрами
  }
}
```

Если в классе определен один или несколько конструкторов с параметрами, нельзя создать объект через неявный конструктор по умолчанию:

```
class Car
{
  private double mileage;
  private double fuel;

  public Car(double mileage, double fuel)
  {
    this.mileage = mileage;
    this.fuel = fuel;
  }
}
class Program
{
  static void Main(string[] args)
  {
    Car newCar = new Car(100, 50);
    Car newCar2 = new Car(); // ошибка, в классе не определен
                             //конструктор без параметров
  }
}
```

Свойства в C#. Аксессоры get и set. Автоматические свойства

Свойство в C# – это член класса, который предоставляет удобный механизм доступа к полю класса (чтение поля и запись). Свойство представляет собой что-то среднее между полем и методом класса. При использовании свойства, мы обращаемся к нему, как к полю класса, но на самом деле компилятор преобразовывает это обращение к вызову соответствующего неявного метода. Такой метод называется аксессор (*accessor*). Существует два таких метода: *get* (для получения данных) и *set*(для записи). Объявление простого свойства имеет следующую структуру:

```
[модификатор доступа] [тип] [имя_свойства]
{
  get
  {
     // тело аксессора для чтения из поля
  }
```

```
  set
  {
     // тело аксессора для записи в поле
  }
}
```

Пример использования свойств. Имеется класс *Студент*, и в нем есть закрытое поле курс, которое не может быть ниже единицы и больше пяти. Для управления доступом к этому полю будет использовано свойство *Year*:

```
class Student
{
  private int year; //объявление закрытого поля

  public int Year //объявление свойства
  {
    get // аксессор чтения поля
    {
       return year;
    }
    set // аксессор записи в поле
    {
        if (value < 1)
          year = 1;
        else if (value > 5)
          year = 5;
        else year = value;
      }
   }
}
class Program
{
  static void Main(string[] args)
  {
     Student st1 = new Student();
     st1.Year = 0; // записываем в поле, используя аксессор  set
    Console.WriteLine(st1.Year); // читаем поле, используя аксессор get,
                                 // выведет 1
    Console.ReadKey();
  }
}
```

В теле аксессора *get* может быть более сложная логика доступа, но в итоге должно возвращаться значение поля, либо другое значение с помощью оператора *return*. В аксессоре *set* же присутствует неявный параметр *value*, который содержит значение, присваиваемое свойству (в примере выше, при записи, значение *value* равно «0»).

```
class Student
{
  private int year;

  public int GetYear()
  {
     return year;
  }
```

```
  public void SetYear(int value)
  {
     if (value < 1)
        year = 1;
     else if (value > 5)
        year = 5;
     else year = value;
  }
}
class Program
{
  static void Main(string[] args)
  {
       Student st1 = new Student();
       st1.SetYear(0);
       Console.WriteLine(st1.GetYear());
       Console.ReadKey();
  }
}
```

Свойство также может предоставлять доступ только на чтение поля или только на запись. Если, например, нам необходимо закрыть доступ на запись, мы просто не указываем аксессор *set*. Пример:

```
class Student
{
  private int year;

  public Student(int y) // конструктор
  {
    year = y;
  }

  public int Year
  {
    get
    {
       return year;
    }
  }
}
class Program
{
  static void Main(string[] args)
  {
     Student st1 = new Student(2);
     Console.WriteLine(st1.Year); // чтение
     st1.Year = 5; // ошибка, свойство только на чтение
     Console.ReadKey();
  }
}
```

Само свойство не определяет место в памяти для хранения поля, и, соответственно, необходимо отдельно объявить поле, доступом к которому будет управлять свойство.

Автоматические свойства

Автоматическое свойство – это очень простое свойство, которое, в отличии от обычного свойства, уже определяет место в памяти (создает неявное поле), но при этом не позволяет создавать логику доступа. Структура объявления Автоматического свойства:

```
[модификатор доступа] [тип] [имя_свойства] { get; set; }
```

У таких свойств, у их аксессоров отсутствует тело. Пример использования:

```
class Student
{
  public int Year { get; set; }
}
class Program
{
  static void Main(string[] args)
  {
      Student st1 = new Student();
      st1.Year = 0;
      Console.WriteLine(st1.Year);
      Console.ReadKey();
  }
}
```

Автоматически реализуемые свойства есть смысл использовать тогда, когда нет необходимости накладывать какие-либо ограничения на возможные значения неявного поля свойства.

И тут у вас может возникнуть вопрос, а в чем тогда разница между простыми открытыми полями и автоматическими свойствами. У таких свойств остается возможность делать их только на чтение или только на запись. Для этого уже используется модификатор доступа *private* перед именем аксессора:

```
public int Year { private get; set; } // свойство только на запись
public int Year { get; private set; } // свойство только на чтение
```

Индексаторы

Индексатор представляет собой разновидность свойства. Он используется для закрытого поля, являющегося массивом.

Ниже приведена общая форма одномерного индексатора:

```
тип_элемента this[int индекс] {
   // Аксессор для получения данных,
   get
  {
      // Возврат значения, которое определяет индекс.
   }
   // Аксессор для установки данных,
   set
  {
      // Установка значения, которое определяет индекс.
   }
}
```

В теле индексатора определены два аксессора (т.е. средства доступа к данным): *get* и *set*. Аксессор подобен методу, за исключением того, что в нем не объявляется тип возвращаемого значения или параметры. Аксессоры вызываются автоматически при использовании индексатора, и оба получают индекс в качестве параметра. Так, если индексатор указывается в левой части оператора присваивания, то вызывается аксессор *set* и устанавливается элемент, на который указывает параметр индекс. В противном случае вызывается аксессор *get* и возвращается значение, соответ-

ствующее параметру индекс. Кроме того, аксессор *set* получает неявный параметр value, содержащий значение, присваиваемое по указанному индексу.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class MyArr
    {
        int[] arr;
        public int Length;

        public MyArr(int Size)
        {
            arr = new int[Size];
            Length = Size;
        }
        // Создаем простейший индексатор
        public int this[int index]
        {
            set
            {
                arr[index] = value;
            }

            get
            {
                return arr[index];
            }
        }
    }

    class Program
    {
        static void Main()
        {
            MyArr arr1 = new MyArr(Size: 5);
            Random ran = new Random();

            // Инициализируем каждый индекс экземпляра класса arr1
            for (int i = 0; i < arr1.Length; i++)
            {
                arr1[i] = ran.Next(1,100);
                Console.Write("{0}\t", arr1[i]);
            }
            Console.ReadLine();
        }
    }
}
```

В текущем классе *MyArr* определен индексатор, позволяющий вызывающему коду идентифицировать подэлементы с применением числовых значений. Однако надо понимать, что это не обязательное требование метода-индексатора.

Индексатор совсем не обязательно должен оперировать массивом. Его основное назначение – предоставить пользователю функциональные возможности, аналогичные массиву.

На применение индексаторов накладываются два существенных ограничения. Во-первых, значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра *ref* или *out*, поскольку в индексаторе не определено место в памяти для его хранения. И во-вторых, индексатор должен быть членом своего класса и поэтому не может быть объявлен как *static*.

Методы

Метод – это небольшая подпрограмма, которая выполняет, в идеале, только одну функцию. Методы позволяют сократить объем кода. Методы вместе с полями, являются основными членами класса.

Статический метод – это метод, который не имеет доступа к полям объекта, и для вызова такого метода не нужно создавать экземпляр (объект) класса, в котором он объявлен.

Простой метод – это метод, который имеет доступ к данным объекта, и его вызов выполняется через объект.

Простые методы служат для обработки внутренних данных объекта.

Приведу пример использования простого метода. Класс *Телевизор*, у него есть поле *switchedOn*, которое отображает состояние включен/выключен, и два метода – включение и выключение:

```
class TVSet
{
    private bool switchedOn;
    public void SwitchOn()
    {
        switchedOn = true;
    }
   public void SwitchOff()
   {
        switchedOn = false;
    }
}
class Program
{
    static void Main(string[] args)
    {
        TVSet myTV = new TVSet();
        myTV.SwitchOn(); // включаем телевизор, switchedOn = true;
        myTV.SwitchOff(); // выключаем телевизор, switchedOn = false;
    }
}
```

Чтобы вызвать простой метод, перед его именем, указывается имя объекта. Для вызова статического метода необходимо указывать имя класса.

Статические методы, обычно, выполняют какую-нибудь глобальную, общую функцию, обрабатывают «внешние данные». Например, сортировка массива, обработка строки, возведение числа в степень и другое.

Пример статического метода, который обрезает строку до указанной длины, и добавляет многоточие:

```
class StringHelper
{
```

```
    public static string TrimIt(string s, int max)
    {
        if (s == null)
            return string.Empty;
        if (s.Length <= max)
           return s;
       return s.Substring(0, max) + "...";
    }
}
class Program
{
    static void Main(string[] args)
    {
       string s = "Очень длинная строка, которую необходимо обрезать до указанной длины
и добавить многоточие";
       Console.WriteLine(StringHelper.TrimIt(s, 20)); //"Очень длинная строка…"
       Console.ReadLine();
    }
}
```

Статический метод не имеет доступа к нестатическим полям класса:

```
class SomeClass
{
    private int a;
    private static int b;

    public static void SomeMethod()
    {
       a=5; // ошибка
       b=10; // допустимо
    }
}
```

Количество и типы параметров вместе с именем метода представляют собой сигнатуру метода. Класс не должен содержат методов с одинаковой сигнатурой. Тип возвращаемого значения в сигнатуру не входит. Все методы класса должны различаться между собой сигнатурой.

Передача параметров в метод по ссылке. Операторы ref и out

В C# значения переменных по-умолчанию передаются по значению (в метод передается локальная копия параметра, который используется при вызове). Это означает, что мы не можем внутри метода изменить параметр из вне:

```
public static void ChangeValue(object a)
{
    a = 2;
}

static void Main(string[] args)
{
    int a = 1;
    ChangeValue(a);
    Console.WriteLine(a); // 1
    Console.ReadLine();
}
```

Чтобы передавать параметры по ссылке, и иметь возможность влиять на внешнюю переменную, используются ключевые слова *ref* и *out*.

Ключевое слово *ref*

Чтобы использовать *ref*, это ключевое слово стоит указать перед типом параметра в методе, и перед параметром при вызове метода:

```
public static void ChangeValue(ref int a)
{
   a = 2;
}
static void Main(string[] args)
{
   int a = 1;
   ChangeValue(ref a);
   Console.WriteLine(a); // 2
   Console.ReadLine();
}
```

В примере изменено значение внешней переменной внутри метода.

Особенностью *ref* является то, что переменная, которая передается в метод, обязательно должна быть проинициализирована значением, иначе компилятор выдаст ошибку

Ключевое слово *out*

Out используется точно таким же образом как и *ref*, за исключением того, что параметр не обязан быть проинициализирован перед передачей, но при этом в методе переданному параметру обязательно должно быть присвоено новое значение:

```
public static void ChangeValue(out int a)
{
   a = 2;
}
static void Main(string[] args)
{
   int a;
   ChangeValue(out a);
   Console.WriteLine(a); // 2
   Console.ReadLine();
}
```

Если не присвоить новое значение параметру out, мы получим ошибку «The out parameter 'a' must be assigned to before control leaves the current method»

Методы с переменным количеством аргументов

В языке C# существует возможность создавать методы, которые можно вызывать с разным количеством параметров. Для этого параметр размещается последним в списке и перед ним указывается ключевое слово params, затем тип, пустые квадратные скобки и имя параметра. Эта запись обозначает массив указанного типа неопределенной длины.

Например, объявление вида:

```
public int summa ( float k, int l, params int[ ] m) {
   // тело
}
```

означает, что в этот метод можно передавать три и более параметра. В теле метода к третьему и последующим параметрам обращаются как к обычным элементам массива. Фактическое количество массива можно получит с помощью его свойства *Length*.

НАСЛЕДОВАНИЕ В C#

В программировании наследование позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником или производным классом. В класс-наследник из базового класса переходят поля, свойства, методы и другие члены класса.

Объявление нового (производного) класса выглядит так:

```
class [имя_класса] : [имя_базового_класса]
{
  // тело класса
}
```

Пример. На основе базового класса *Животное* создаются два класса *Собака* и *Кошка*, в эти два класса переходит свойство *ИмяЖивотного*:

```
class Animal
{
  public string Name { get; set; }
}
class Dog : Animal
{
  public void Guard()
  {
    // собака охраняет
  }
}
class Cat : Animal
{
  public void CatchMouse()
  {
    // кошка ловит мышь
  }
}

class Program
{
 static void Main(string[] args)
  {
    Dog dog1 = new Dog();
    dog1.Name = "Барбос"; // называем пса
    Cat cat1 = new Cat();
    cat1.Name = "Барсик"; // называем кота
    dog1.Guard(); // отправляем пса охранять
    cat1.CatchMouse(); // отправляем кота на охоту
  }
}
```

Вызов конструктора базового класса в C#

Когда конструкторы объявлены и в базовом классе, и в наследнике –необходимо вызывать их оба. Для вызова конструктора базового класса используется ключевое слово *base*. Объявление

конструктора класса-наследника с вызовом базового конструктора имеет следующую структуру:

```
[имя_конструктора_класса-наследника] ([аргументы]) : base ([аргументы])
{
  // тело конструктора
}
```

В базовый конструктор передаются все необходимые аргументы для создания базовой части объекта.

```
class Animal
{
  public string Name { get; set; }

  public Animal(string name)
  {
    Name = name;
  }
}
class Parrot : Animal
{
  public double BeakLength { get; set; } // длина клюва

  public Parrot(string name, double beak) : base(name)
  {
    BeakLength = beak;
  }
}
class Dog : Animal
{
 public Dog(string name) : base (name)
  {
    // здесь может быть логика создания объекта Собака
  }
}
class Program
{
  static void Main(string[] args)
 {
    Parrot parrot1 = new Parrot("Кеша", 4.2);
    Dog dog1 = new Dog("Барбос");
  }
}
```

Доступ к членам базового класса из класса-наследника

В классе-наследнике можно получить доступ к членам базового класса которые объявлены как *public*, *protected*, *internal* и *protected internal*. Члены базового класса с модификатором доступа *private* также переходят в класс-наследник, но к ним могут иметь доступ только члены базового класса. Например, свойство, объявленное в базовом классе, которое управляет доступом к закрытому полю, будет работать корректно в классе-наследнике, но отдельно получить доступ к этому полю из класса-наследника мы не сможем.

Перегрузка операторов

Перегрузка оператора - это реализация своего собственного функционала этого оператора

для конкретного класса.
Перегрузка унарного оператора:

```
public static [возвращаемый_тип] operator [оператор]([тип_операнда] [операнд])
{
  //функционал оператора
}
```

Перегрузка бинарного оператора:

```
public static [возвращаемый_тип] operator [оператор]([тип_операнда1] [операнд1],
[тип_операнда2] [операнд2])
{
  //функционал оператора
}
```

Модификаторы *public* и *static* являются обязательными. На месте [оператор] может стоять любой оператор, который можно перегрузить. Не все операторы в C# разрешается перегружать.

Можно перегружать:

унарные операторы: +, -, !, ++, —, true, false

бинарные операторы: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=

Нельзя перегружать

[] – функционал этого оператора предоставляют индексаторы

() – функционал этого оператора предоставляют методы преобразования типов

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= краткие формы оператора присваивания будут автоматически доступны при перегрузке соответствующих операторов (+, -, * …).

Пример перегрузки бинарных операторов

```
public class Money
{
  public decimal Amount { get; set; }
  public string Unit { get; set; }

  public Money(decimal amount, string unit)
  {
    Amount = amount;
    Unit = unit;
  }
  public static Money operator +(Money a, Money b) //перегрузка оператора «+»
  {
    if (a.Unit != b.Unit)
      throw new InvalidOperationException("Нельзя суммировать деньги в разных
валютах");

    return new Money(a.Amount + b.Amount, a.Unit);
  }

}
class Program
{
  static void Main(string[] args)
```

```
  {
    Money myMoney = new Money(100, "USD");
    Money yourMoney = new Money(100, "RUR");
    Money hisMoney = new Money(50, "USD");
    Money sum = myMoney + hisMoney; // 150 USD
    sum = yourMoney + hisMoney; // исключение - разные валюты
    Console.ReadLine();
  }
}
```

Перегрузка унарных операторов

Унарные операторы «++», «--» (добавление/вычитание 1 единицы денег):

```
public class Money
{
  public decimal Amount { get; set; }
  public string Unit { get; set; }

  public Money(decimal amount, string unit)
  {
    Amount = amount;
    Unit = unit;
  }
  public static Money operator +(Money a, Money b)
  {
    if (a.Unit != b.Unit)
      throw new InvalidOperationException("Нельзя суммировать деньги в разных
валютах");
    return new Money(a.Amount + b.Amount, a.Unit);
  }
  public static Money operator ++(Money a) // перегрузка «++»
  {
    a.Amount++;
    return a;
  }
  public static Money operator --(Money a) // перегрузка «--»
  {
    a.Amount--;
    return a;
  }

}
class Program
{
  static void Main(string[] args)
  {
    Money myMoney = new Money(100, "USD");
    myMoney++; // 101 USD
    Console.ReadLine();
  }
}
```

Также существует возможность перегрузки самого операторного метода. Это означает, что

в классе может быть несколько перегрузок одного оператора при условии что входные параметры будут отличаться типом данных (например, когда необходимо сложить объект класса и строку):

```
public class Money
{
  public decimal Amount { get; set; }
  public string Unit { get; set; }

  public Money(decimal amount, string unit)
  {
    Amount = amount;
    Unit = unit;
  }
  public static Money operator +(Money a, Money b)
 {
    if (a.Unit != b.Unit)
      throw new InvalidOperationException("Нельзя суммировать деньги в разных
валютах");

     return new Money(a.Amount + b.Amount, a.Unit);
  }
  public static string operator +(string text, Money a)
  {
     return text + a.Amount + " " + a.Unit;
  }

}
class Program
{
  static void Main(string[] args)
  {
      Money myMoney = new Money(100, "USD");
      Console.WriteLine("У меня сейчас " + myMoney); // "У меня сейчас 100 USD"
     Console.ReadLine();
  }
}
```

Виртуальные методы

Виртуальный метод – это метод, который может быть переопределен в классе наследнике. Переопределение метода – это изменение его реализации в классе наследнике. Переопределив метод, он будет работать по-разному в базовом классе и классе наследнике, имея при этом одно и то же имя и аргументы .

```
 [модификатор доступа] virtual [тип] [имя метода] ([аргументы])
{
  // тело метода
}
```

Статический метод не может быть виртуальным.

Объявив виртуальный метод, можно переопределить его в классе наследнике.

```
[модификатор доступа] override [тип] [имя метода] ([аргументы])
{
  // новое тело метода
}
```

Пример. Базовый класс - Человек, производные – *Студент* и *Ученик*. В базовом классе есть виртуальный метод *ShowInfo*, который выводит информацию об объекте. В классах *Студент* и *Ученик* этот метод переопределяется.

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

  public Person(string name, int age)
  {
    Name = name;
    Age = age;
  }
  public virtual void ShowInfo() //объявление виртуального метода
  {
    Console.WriteLine("Человек\nИмя: " + Name + "\n" + "Возраст: " + Age + "\n");
  }
}
class Student : Person
{
  public string HighSchoolName { get; set; }

  public Student(string name, int age, string hsName): base(name, age)
  {
    HighSchoolName = hsName;
  }
  public override void ShowInfo() // переопределение метода
  {
    Console.WriteLine("Студент\nИмя: " + Name + "\n" + "Возраст: " + Age +"\n"+ "Название
ВУЗа: " + HighSchoolName + "\n");
  }
}
class Pupil : Person
{
  public string Form { get; set; }

  public Pupil(string name, int age, string form): base(name, age)
  {
    Form = form;
  }
  public override void ShowInfo() // переопределение метода
  {
    Console.WriteLine("Ученик(ца)\nИмя: " + Name + "\n" + "Возраст: " + Age + "\n" + "Класс: "
+ Form + "\n");
  }
}

class Program
{

  static void Main(string[] args)
```

```
  {
    List<Person> persons = new List<Person>();
    persons.Add(new Person("Василий", 32));
    persons.Add(new Student("Андрей", 21, "МГУ"));
    persons.Add(new Pupil("Елена", 12, "7-Б"));

    foreach (Person p in persons)
       p.ShowInfo();
  }
}
```

Если убрать переопределение, откинув ключевые слова *virtual* и *override* в базовом классе, в классе наследнике будут методы с одинаковым именем *ShowInfo*. Программа работать будет, но о каждом объекте, независимо это просто человек или студент/ученик, будет выводиться информация только как о простом человеке (будет вызываться метод *ShowInfo* из базового класса).

Это можно исправить, добавив проверки на тип объекта, и при помощи приведения типов, вызывать нужный метод *ShowInfo*:

```
foreach (Person p in persons)
{
  if (p is Student)
    ((Student)p).ShowInfo();
  else if (p is Pupil)
    ((Pupil)p).ShowInfo();
  else p.ShowInfo();
}
```

Событийно-управляемое программирование

В основу ОС Windows положен принцип *событийного управления.* Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется *событием.*

Событие воспринимается Windows и преобразуется в *сообщение* запись, содержащую необходимую информацию о событии (например, какая клавиша была нажата, в каком месте экрана произошел щелчок мышью). Сообщения могут поступать не только от пользователя, но и от самой системы, а также от активного или других приложений. Определен достаточно широкий круг стандартных сообщений, образующий иерархию, кроме того, можно определят собственные сообщения.

Сообщения поступают в общую очередь, откуда распределяются по очередям приложений. Каждое приложение содержит *цикл обработки сообщений*, которое выбирает сообщение из очереди и через операционную систему вызывает подпрограмму, предназначенную для его обработки. Таким образом, Windows-приложение состоит из главной программы, содержащей цикл обработки сообщений, инициализацию и завершение приложения, и набора *обработчиков событий.*

Среда Visual Studio.NET содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу создание шаблонов и заготовок обработчиков событий.

Шаблон Windows-приложения

Для создания нового проекта (File ► New ► Project), нужно выбрать шаблон Windows Application.

Среда самостоятельно сформирует шаблон Windows-приложения.Среда создает не только заготовку формы, но и шаблон текста приложения. Перейти к нему можно, щелкнув в окне Solu-

tion Explorer (View ► Solution Explorer) правой кнопкой мыши на файле Forml.cs и выбрав в контекстном меню команду View Code. При этом откроется вкладка с кодом формы вида:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace WindowsApplicationl
{
   public class Forml : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components = null;
      public Form()
      {
         InitializeComponent( );
      }
      protected override void Dispose (bool disposing )
      {
         if (disposing )
         {
            if (components != null)
            {
               components.Dispose( );
            }
         }
         base.Dispose (disposing );
      }

      #region Windows Form Designer generated code
      private void InitializeComponent( )
      {
         this.components = new System.ComponentModel.Container();
         this.Size = new System.Drawing.Size(300.300);
         this.Text = "Forml";
      }
      #endregion

      static void Main()
      {
         Application.Run(new Forml( ));
      }
   }
}
```

Приложение начинается с директив использования пространств имен библиотеки .NET. Для пу-

стой формы, не содержащей ни одного компонента, необходимыми являются только две директивы:

```
using System;
using System.Windows.Forms;
```

Список наиболее употребительных элементов этого пространства имен:

Application	Класс Windows-приложения. При помощи методов этого класса можно обрабатывать Windows-сообщения, запускать и прекращать работу приложения и т. п.
ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox	Примеры классов, представляющих элементы управления (компоненты): базовый класс кнопок, кнопка, флажок, комбинированный-список, таблица, группа, список, метка с гиперссылкой, изображение.
Form	Класс формы - окно Windows-приложения.
ColorDialog, FileDialog, PrintPreviewDialog, FontDialog	Примеры стандартных диалоговых окон для выбора цветов, файлов, шрифтов, окно предварительного просмотра.
Menu, MainMenu, Menultem, ContextMenu	Классы выпадающих и контекстных меню.
Clipboard, Help, ToolTip, Cursors	Вспомогательные типы для организации графических Timer, Screen, интерфейсов: буфер обмена, помощь, таймер, экран, подсказка, указатели мыши
StatusBar, Splitter, Bar, Tool Scroll Bar	Примеры дополнительных элементов управления, размещаемых на форме: строка состояния, разделитель, панель инструментов и т. д.

Класс *Form1* является производным от класса *Form*. Он наследует от своего предка множество элементов. В самом классе *Form1* описано новое закрытое поле *components* - контейнер для хранения компонентов, которые можно добавить в класс формы.

Конструктор формы вызывает закрытый метод *InitializeComponent*, автоматически формируемый средой (код метода скрыт между директивами препроцессора *#region* и *#endregion*). Этот метод обновляется средой при добавлении элементов управления на форму, а также при изменении свойств формы и содержащихся на ней элементов. Например, если изменить цвет фона формы с помощью окна свойств (*Properties*), в методе появится примерно такая строка:

```
this.BackColor = System.Drawing.SystemColors.AppWorkspace;
```

Метод освобождения ресурсов *Dispose* вызывается автоматически при закрытии формы. Наиболее часто используемые события:

Activated	получение формой фокуса ввода
Click, Doubleclick	одинарный и двойной щелчки мышью
Closed	закрытие формы
Load	загрузка формы
KeyDown, Keyllp	нажатие и отпускание любой клавиши и их сочетании
KeyPress	нажатие клавиши, имеющей ASCII-код
MouseDown, MouseUp	нажатие и отпускание кнопки мыши
MouseMove	перемещение мыши
Paint	возникает при необходимости прорисовки формы

Имя обработчика формируется средой автоматически из имени экземпляра компонента и имени события. Обратите внимание на то, что обработчикам передаются два параметра, объект-источник события и запись, соответствующая типу события. При задании обработчика можно задать и другое имя, для этого оно записывается справа от имени соответствующего события на вкладке *Events* окна свойств

Класс Control

Класс *Control* является базовым для всех отображаемых элементов, которые составляют графический интерфейс пользователя, кнопок, списков, полей ввода и форм. Класс *Control* реализует базовую функциональность интерфейсных элементов. Он содержит методы обработки ввода пользователя с помощью мыши и клавиатуры, определяет размер, положение, цвет фона и другие характеристики элемента. Для каждого объекта можно определить родительский класс, задав свойство *Parent*, при этом объект будет иметь, например, такой же цвет фона, как и его родитель.

Наиболее важные свойства класса *Control*:

Свойство	Описание
Anchor	Определяет, какие края элемента управления будут при вязаны к краям родительского контейнера. Если задать привязку всех краев, элемент будет изменять размеры вместе с родительским
BackColor, Backgroundlmage, Font, ForeColor, Cursor	Определяют параметры отображения рабочей области формы: цвет фона, фоновый рисунок, шрифт, цвет текста, вид указателя мыши
Bottom, Right	Координаты нижнего правого угла элемента. Могут устанавливаться также через свойство Size
Top, Left	Координаты верхнего левого угла элемента. Эквивалентны свойству Location
Bounds	Возвращает объект типа Rectangle (прямоугольник), который определяет размеры элемента управления
ClientRectangle	Возвращает объект Rectangle, определяющий размеры рабочей области элемента
ContextMenu	Определяет, какое контекстное меню будет выводиться при щелчке на элементе правой кнопкой мыши
Dock	Определяет, у какого края родительского контейнера будет отображаться элемент управления
Location	Координаты верхнего левого угла элемента относительно верхнего левого угла контейнера, содержащего этот элемент, в виде структуры типа Point. Структура содержит свойства X и У
Height, Width	Высота и ширина элемента
Size	Высота и ширина элемента в виде структуры типа Size. Структура содержит свойства Height и Width
treated, Disposed, Enabled, Focused, Visible	Возвращают значения типа bool, определяющие текущее состояние элемента: создан, удален, использование разрешено, имеет фокус ввода, видимый
Handle	Возвращает дескриптор элемента (уникальное целочисленное значение, сопоставленное элементу)
ModifierKeys	Статическое свойство, используемое для проверки состояния модифицирующих клавиш (Shift, Control, Alt). Возвращает результат в виде объекта типа Keys
MouseButtons	Статическое свойство, проверяющее состояние клавиш мыши. Возвращает результат в виде объекта типа MouseButtons
Opacity	Определяет степень прозрачности элемента управления. Может изменяться от 0 (прозрачный) до 1 (непрозрачный)
Parent	Возвращает объект, родительский по отношению к данному (имеется в виду не базовый класс, а объект-владелец)
Region	Определяет объект Region, при помощи которого можно управлять очертаниями и границами элемента управления

Элементы управления

Элементы управления, или *компоненты,* помещают на форму с помощью панели инструментов ToolBox (View ► ToolBox). В этом разделе кратко описаны простейшие компоненты и

приведены примеры их использования.

Метка Label

Метка предназначена для размещения текста на форме. Текст хранится в свойстве *text*. Можно задавать шрифт (свойство *Font*), цвет фона (*BackColor*), цвет шрифта (*ForeColor*) и выравнивание (*TextAlign*) текста метки. Метка может автоматически изменять размер в зависимости от длины текста (*AutoSize* = True). Можно разместить на метке изображение (*Image*) и задать прозрачность (установить для свойства *BackColor* значение *Color*. *Transparent*).

Кнопка Button

Основное событие, обрабатываемое кнопкой, - щелчок мышью (*Click*). Кроме того, кнопка может реагировать на множество других событий - нажатие клавиш на клавиатуре и мыши, изменение параметров и т. д.

Поле ввода TextBox

Компонент TextBox позволяет пользователю вводить и редактировать текст, который запоминается в свойстве Text. Можно вводить строки практически неограниченной длины (приблизительно до 32 000 символов), корректировать их, а также вводить защищенный текст (пароль) путем установки маски, отображаемой вместо вводимых символов (свойство *PasswordChar*). Для обеспечения возможности ввода нескольких строк устанавливают свойства *Multiline*, *ScrollBars* и *Wordwrap*. Доступ только для чтения устанавливается с помощью свойства *Readonly*. Элемент содержит методы очистки (Clear), выделения (Select), копирования в буфер (Copy), вставки из него (Paste) и др., а также реагирует на множество событий, основными из которых являются Keypress и KeyDown.

Меню MainMenu и ContextMenu

Главное меню размещается на форме двойным щелчком на его значении на панели Toolbox.

Каждый пункт меню – объект типа MenuItem. При вводе пункта меню задается свойство Text. Переход к заданию следующего пункта меню выполняется щелчком мыши.

Пункт меню может быть разрешен или запрещен, видимым или невидимым, отмечен или не отмечен. Заготовка обработчика событий формируется двойным щелчком на пункте меню.

Любое приложение завершается командой Exit или методом Close класса главной формы.

Контекстное меню вызывается во время выполнения программы по нажатию правой кнопки мыши на форме или элементе управления. Обычно в этом меню размещаются пункты, дублирующие пункты главного меню, или пункты, определяющие специфические для данного компонента действия.

Контекстное меню ContextMenu создается и используется аналогично главному (значок контекстного меню появляется на панели инструментов, если воспользоваться кнопкой прокрутки). Для привязки контекстного меню к компоненту следует установить значение свойства ContextMenu этого компонента равным имени контекстного меню.

Флажок CheckBox

Флажок используется для включения-выключения пользователем какого-либо режима. Для проверки, установлен ли флажок, анализируют его свойство Checked, принимающее значение true или false. Флажок может иметь и третье состояние — «установлен, но не полностью». Как правило, это происходит в тех случаях, когда устанавливаемый режим определяется несколькими «подрежимами», часть из которых включена, а часть выключена. В этом случае используют свойство *CheckState*, которое может принимать значения *Checked*, *Unchecked* и *Intermediate*. Кроме того, флажок обладает свойством *ThreeState*, которое управляет возможностью установки третьего состояния пользователем с помощью мыши. Для флажка можно задать цвет фона и фоновое изображение так же, как и для метки. Свойство Appearance управляет отображением флажка: либо в виде

собственно флажка (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью.

Флажки используются в диалоговых окнах как поодиночке, так и в группе, причем все флажки устанавливаются независимо друг от друга.

Переключатель RadioButton

Переключатель позволяет пользователю выбрать один из нескольких предложенных вариантов, поэтому переключатели обычно объединяют в группы. Если один из них устанавливается (свойство Checked), остальные автоматически сбрасываются. Программист может менять стиль и цвет текста, связанного с переключателем, и его выравнивание. Для переключателя можно задать цвет фона и фоновое изображение так же, как и для метки.

Переключатели можно поместить непосредственно на форму, в этом случае все они составят одну группу. Если на форме требуется отобразить несколько групп переключателей, их размещают внутри компонента Group или Panel.

Свойство Appearance управляет отображением переключателя: либо в традиционном виде (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью.

Панель GroupBox

Панель GroupBox служит для группировки элементов на форме, например для того, чтобы дать общее название и визуально выделить несколько переключателей или флажков, обеспечивающих выбор связанных между собой режимов.

Список ListBox

Список служит для представления перечней элементов, в которых пользователь может выбрать одно (свойство SelectionMode равно One) или несколько значений (свойство SelectionMode равно MultiSimple или Multi Extended). Если значение свойства равно MultiSiple, щелчок мышью на элементе выделяет или снимает выделение. Значение MultlExtended позволяет использовать при выделении диапазона строк клавишу Shift, а при добавлении элемента - клавишу Ctrl. аналогично проводнику Windows. Запретить выделение можно установив значение свойства SelectionMode, равное None.

Чаще всего используются списки строк, но можно выводить и произвольные изображения. Список может состоять из нескольких столбцов (свойство MultiColumn) быть отсортированным в алфавитном порядке (Sorted = True).

Элементы списка нумеруются с нуля. Они хранятся в свойстве Items, представляющую собой коллекцию.

Формы

Окна приложения могут иметь различные вид и назначение. Все окна можно разделить на модальные и немодальные. *Модальное окно* не позволяет пользователю переключаться на другие окна того же приложения, пока не будет завершена работа с текущим окном. В виде модальных обычно оформляют *диалоговые окна*, требующие от пользователя ввода какой-либо информации. Модальное окно можно закрыть щелчком на кнопке наподобие ОК, подтверждающей введенную информацию, на кнопке закрытия окна или на кнопке вроде Cancel, отменяющей ввод пользователя.

Немодальное окно позволяет переключаться на другие окна того же приложения. Немодальные окна являются, как правило, информационными. Они используются в тех случаях, когда пользователю желательно предоставить свободу выбора – оставлять на экране какую-либо информацию или нет.

Каждое приложение содержит одно *главное окно.* Класс главного окна приложения содержит точку входа в приложение (статический метод Main). При закрытии главного окна приложение завершается.

В случае использования *многодокументного интерфейса* (Multiple Document Interface,

MDI) одно *родительское окно* может содержать другие окна, называемые *дочерними.* При закрытии родительского окна дочерние окна закрываются автоматически. Вид окна определяет его функциональность, например, окно с одинарной рамкой не может изменять свои размеры.

Класс Form

Класс Form представляет собой заготовку формы, от которой наследуются классы форм приложения. Помимо множества унаследованных элементов, в этом классе определено большое количество собственных элементов.

Свойство	Описание
AcceptButton	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Enter
ActiveMDIChild, IsMDIChild, IsMDIContainer	Свойства предназначены для использования в приложениях с многодокументным интерфейсом (MDI)
AutoScale	Позволяет установить или получить значение, определяющее, будет ли форма автоматически изменять свои размеры, чтобы соответствовать высоте, используемого на форме, или размерам размещенных на ней компонентов
FormBorderStyle	Позволяет установить или получить стиль рамки вокруг формы (используются значения перечисления
Cancel Button	Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Esc
Control Box	Позволяет установить или получить значение, определяющее, будет ли присутствовать стандартная кнопка системного меню в верхнем левом углу заголовка формы
Menu, MergedMenu	Используются для установки или получения информации о меню на форме
MaximizeBox, MinimizedBox	Определяют, будут ли на форме присутствовать стандартные кнопки восстановления и свертывания в правом верхнем углу заголовка формы

Диалоговые окна

В библиотеке .NET нет специального класса для представления диалоговых окон. Вместо этого устанавливаются определенные значения свойств в классе обычной формы. В диалоговом окне можно располагать те же элементы управления, что и на обычной форме. *Диалоговое окно характеризуется:*

неизменяемыми размерами (FormBorderStyle = FixedDialog);

отсутствием кнопок восстановления и свертывания в правом верхнем углу а головка формы (MaximizeBox = False, MinimizedBox = False);

наличием кнопок наподобие ОК, подтверждающей введенную информацию, и Cancel, отменяющей ввод пользователя, при нажатии которых окно закрывается (AcceptButton = имя_кногжи_ОК, Cancel Button = имя_кнопки_Cance1);

установленным значением свойства DialogResult для кнопок, при нажатии которых окно закрывается.

Для отображения диалогового окна используется метод ShowDialog, который формирует результат выполнения из значений перечисления DialogResult. Если пользователь закрыл диалоговое окно щелчком на кнопке наподобие ОК, введенную им информацию можно использовать в дальнейшей работе. Закрытие окна щелчком на кнопке вроде Cancel отменяет все введенные данные. Диалоговое окно обычно появляется при выборе пользователем некоторой команды меню на главной форме.

Если пользователь введет текст в диалоговое окно и закроет его щелчком на кнопке ОК,

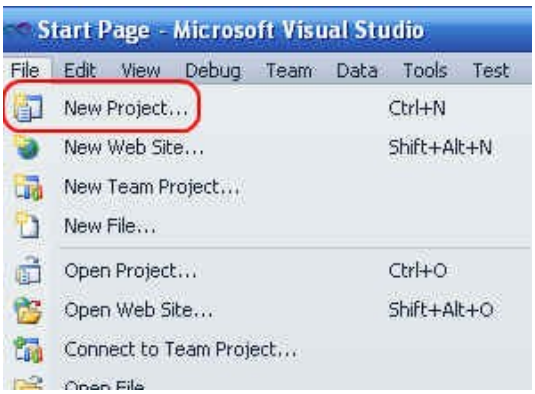
этот текст отобразится в поле метки в главном окне. Если диалоговое окно будет закрыто щелчком на кнопке Cancel, поле метки останется неизменным.

Класс Application

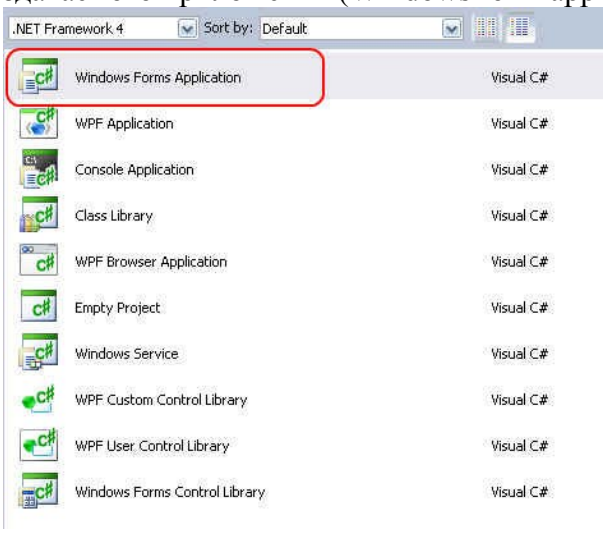
Класс Application, описанный в пространстве имен System.Windows.Forms, содержит статические свойства, методы и события, предназначенные для управления приложением в целом и получения его общих характеристик. Наиболее важные элементы класса Application действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс.

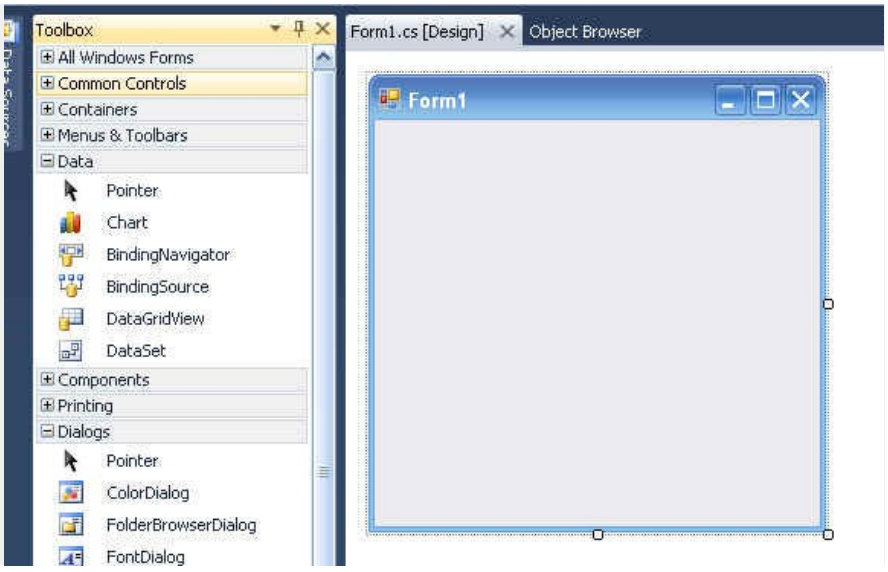
Элемент класса	Тип	Описание
AddMessageFilter RemoveMessageFiIter	Методы	Позволяют перехватывать сообщения и выполнять с ними нужные предварительные действия.
DoEvents	Метод	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции
Exit	Метод	Завершает работу приложения
ExitThread	Метод	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
Run	Метод	Запускает стандартный цикл обработки сообщений для текущего потока
CommonAppDataRegistry	Свойство	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Свойство	Возвращает имя компании
CurrentCulture	Свойство	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentlnputLanguage	Свойство	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Свойство	Позволяет получить имя программного продукта, которое ассоциировано с данным приложением
ProductVersion	Свойство	Позволяет получить номер версии программного продукта
StartupPath	Свойство	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе
ApplicationExit	Событие	Возникает при завершении приложения
Idle	Событие	Возникает, когда все текущие сообщения в очереди обработаны и приложение переходит в режим бездействия
ThreadExit	Событие	Возникает при завершении работы потока в приложении. Если работу завершает главный поток приложения, это событие возникает до события ApplicationExit

Создание приложения Windows Form в Microsoft Visual Studio . Создаем проект через пункт меню "File" ► "New Project":

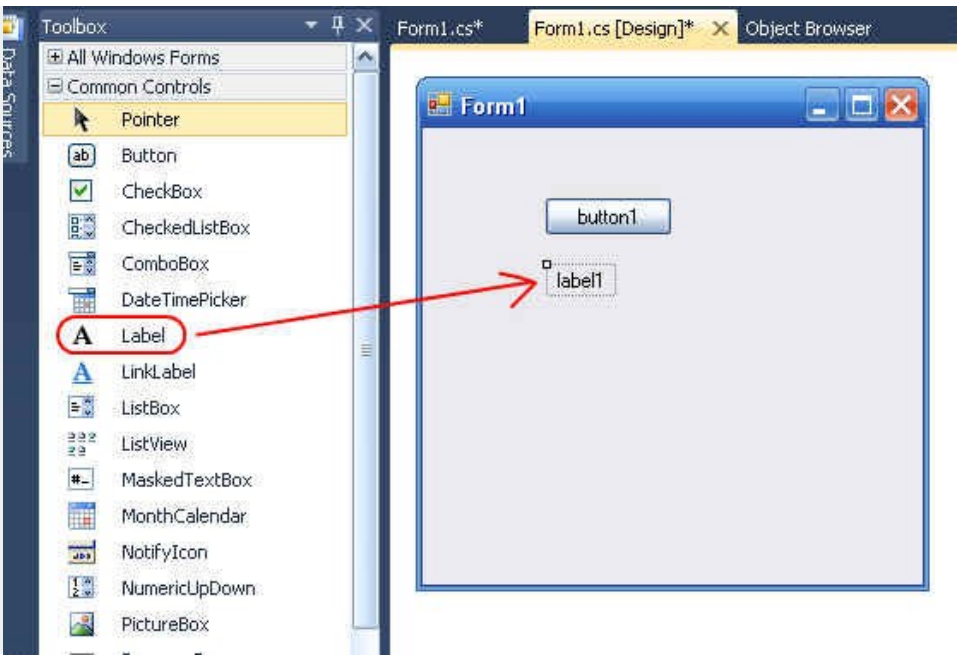


Теперь надо выбрать тип создаваемого приложения (Windows form application):



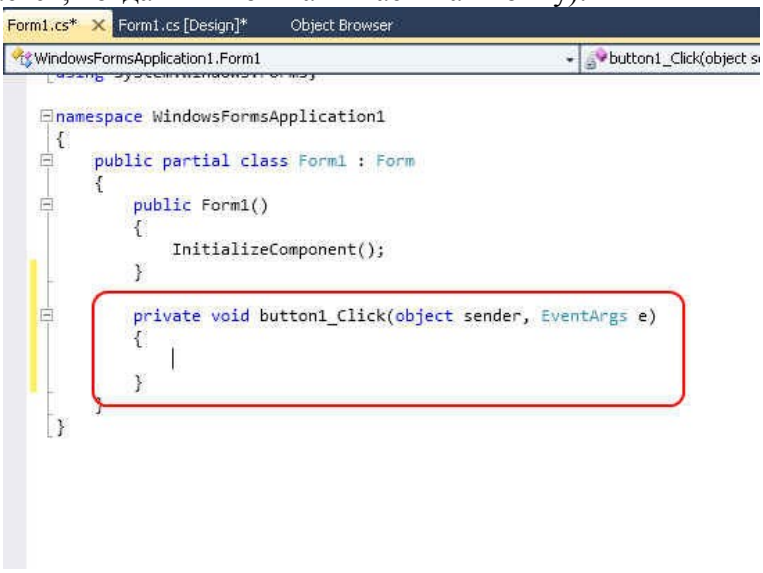


Слева находится панель инструментов, справа шаблон формы



Добавляем на форму кнопку и надпись

Теперь щелкнем на положенную нами на форму кнопку. У нас откроется окно редактирования кода, при этом будет еще и автоматически создан шаблон обработчика нажатия кнопочки (участка кода, который запускается, когда мышкой нажимаем на кнопку):



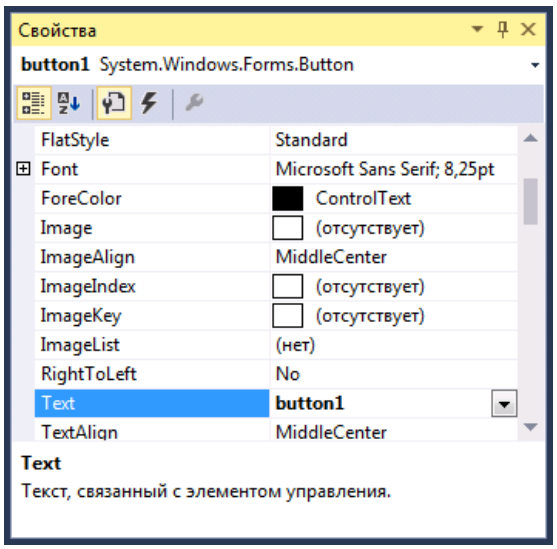
В выделенное поле вставляем данный код

```
label1.Text = "Hello, world!";
```

Для того чтобы сделать кнопку для открытия нового окна, необходимо добавить новую форму, после чего добавить на первую форму кнопку, по нажатию которой будет выводиться вторая форма.

```
Form2 frm = new Form2(this);
frm.ShowDialog();
```

Для изменения текста надписей для элементов Label и Button . необходимо элемент label1, перейдите в Панель свойств, (обычно она находится под Панелью элементов, если панель выключена, включите ее в меню Вид -> Диспетчер свойств.) и задайте для данного элемента значение атрибута Text как показано на рисунке.



Передача данных из одной формы в другую

Часто возникает необходимость передать данные из одной формы в другую, поэтому рассмотрим различные способы и их недостатки и достоинства.

Изменение модификатора доступа
В Form2 Установить модификатор доступа для контрола/поля public.
В любом месте Form1написать:

```
Form2 f = new Form2();
f.ShowDialog();
this.textBox1.Text = f.textBox1.Text;
```

+ Самый быстрый в реализации и удобный способ;
- Противоречит всем основам ООП;
- Возможна передача только из более поздней формы в более раннюю;
- Форма f показывается только с использованием ShowDialog(), т.е. в первую форму управление вернется только по закрытию второй. Избежать этого можно, сохранив ссылку на вторую форму в поле первой формы.

Использование открытого свойства/метода
В классе Form2 определяем свойство (или метод):

```
public string Data
{
   get
   {
      return textBox1.Text;
   }
}
```

В любом месте Form1:

```
Form2 f = new Form2();
f.ShowDialog();
this.textBox1.Text = f.Data;
```

+ Противоречит не всем основам ООП;
- Минусы те же.

Передача данных в конструктор Form2

Изменяем конструктор Form2:

```
public Form2(string data)
{
    InitializeComponent();
    //Обрабатываем данные
    //Или записываем их в поле
    this.data = data;
}
string data;
```

А создаем форму в любом месте Form1 так:

```
Form2 f = new Form2(this.textBox1.Text);
f.ShowDialog();
//Или f.Show();
```

+ Простой в реализации способ;
+ Не нарушает ООП;
- Возможна передача только из более ранней формы в более позднюю.

Передача ссылки в конструктор
Изменяем конструктор Form2:

```
public Form2(Form1 f1)
{
    InitializeComponent();
    //Обрабатываем данные
    //Или записываем их в поле
    string s = f1.textBox1.Text;
}
```

А создаем форму в любом месте Form1 так, т.е. передаем ей ссылку на первую форму:

```
Form2 f = new Form2(this);
f.ShowDialog();
//Или f.Show();
```

+ Доступ ко всем открытым полям/функциям первой формы;
+ Передача данных возможна в обе стороны;
- Нарушает ООП.

Используем свойство 'родитель'
При создании второй формы устанавливаем владельца:

```
Form2 f = new Form2();
f.Owner = this;
f.ShowDialog();
```

Во второй форме определяем владельца:

```
Form1 main = this.Owner as Form1;
if(main != null)
{
    string s = main.textBox1.Text;
    main.textBox1.Text = "OK";
}
```

+ Доступ ко всем открытым полям/функциям первой формы;

+ Передача данных возможна в обе стороны;
+ Не нарушает ООП;

Используем отдельный класс
Создаем отдельный класс, лучше статический, в основном namespace, т.е., например, в файле Program.cs:

```
static class Data
{
    public static string Value { get; set; }
}
```

Его открытые свойства/методы доступны из любой формы:

```
Data.Value = "111";
```

+ Самый удобный способ, когда данные активно используются несколькими формами.

Обработка исключительных ситуаций в C#

Язык C# содержит операторы, позволяющие обнаруживать и обрабатывать ошибки, так называемые исключительные ситуации, возникающие в процессе выполнения программы.

Исключительная ситуация (исключение) – это возникновение аварийного состояние, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение.

Исключения в C# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например нажатие определенной клавиши.

Исключение позволяют логически разделить вычислительный процесс на две части: обнаружение аварийной ситуации и ее обработка. Функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место ее возникновения. Это особенно актуально при использовании библиотечных функций н программ, состоящих из многих модулей.

Другим достоинством исключений является то, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение и параметры, поэтому заголовки функций не разрастаются.

Исключения обнаруживаются и обрабатываются в операторе *try*. Часто используемыми стандартными исключениями являются:

ArithmeticException	Ошибка в арифметических операциях или преобразованиях
ArrayTypeismatchException	Попытка сохранения в массиве элемента несовместимого типа
DivideByZeroException	Попытка деления на ноль
FormatException	Попытка передать в метод аргумент неверного формата
IndexOut0fRangeException	Индекс массива выходит за границы диапазона
InvalidCastException	Ошибка преобразования типа
Out0fMemoryException	Недостаточно памяти для создания нового объекта
OverFlowException	Переполнение при выполнении арифметических операций
StackOverFlowException	Переполнение стека

Оператор try-catch

Обработка исключений – это описание реакции программы на подобные события (исключения) во время выполнения программы. Реакцией программы может быть корректное завершение работы программы, вывод информации об ошибке и запрос повторения действия (при вводе данных).

Примерами исключений может быть:

- деление на ноль;
- конвертация некорректных данных из одного типа в другой;
- попытка открыть файл, которого не существует;
- доступ к элементу вне рамок массива;
- исчерпывание памяти программы;
- другое.

Для обработки исключений в C# используется оператор *try-catch*. Он имеет следующую структуру:

```
try
{
   //блок кода, в котором возможно исключение
}
catch ([тип исключения] [имя])
{
   //блок кода – обработка исключения
}
```

Выполняется код в блоке *try*, и, если в нем происходит исключение типа, соответствующего типу, указанному в *catch*, то управление передается блоку *catch*. При этом, весь оставшийся код от момента выбрасывания исключения до конца блока *try* не будет выполнен. После выполнения блока *catch*, оператор *try-catch* завершает работу.

Указывать имя исключения не обязательно. Исключение представляет собою объект, и к нему мы имеем доступ через это имя. С этого объекта мы можем получить, например, стандартное сообщение об ошибке (*Message*), или трассировку стека (*StackTrace*), которая поможет узнать место возникновения ошибки. В этом объекте хранится детальная информации об исключении.

Если тип выброшенного исключения не будет соответствовать типу, указанному в *catch* – исключение не обработается, и программа завершит работу аварийно.

Ниже приведен пример программы, в которой используется обработка исключения некорректного формата данных:

```
static void Main(string[] args)
{
  string result = "";
  Console.WriteLine("Введите число:");
  try
  {
   int a = Convert.ToInt32(Console.ReadLine()); //вводим данные,
                                                //и конвертируем в целое число
   result = "Вы ввели число " + a;
  }
  catch (FormatException)
  {
   result = "Ошибка. Вы ввели не число";
  }
  Console.WriteLine(result);
  Console.ReadLine();
}
```

Одному блоку *try* может соответствовать несколько блоков catch:

В зависимости от того или другого типа исключения в блоке *try*, выполнение будет передано соответствующему блоку *catch*.

Оператор *try-catch* также может содержать блок *finally*. Особенность блока *finally* в том, что код внутри этого блока выполнится в любом случае, в независимости от того, было ли исключение

или нет.

Выполнение кода программы в блоке *finally* происходит в последнюю очередь. Сначала *try* затем *finally* или *catch-finally* (если было исключение).

Обычно, он используется для освобождения ресурсов. Классическим примером использования блока *finally* является закрытие файла. *Finally* гарантирует выполнение кода, несмотря ни на что. Даже если в блоках *try* или *catch* будет происходить выход из метода с помощью оператора *return – finally* выполнится.

Операторы *try-catch* также могут быть вложенными. Внутри блока try либо *catch* может быть еще один *try-catch*. Генерация исключения выполняется с помощью оператора *throw*.

Оператор *throw* с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного класса *System Exception*. Этот объект используется для передачи информации об исключении – обработчику.

Пример:

```
using System;
namespace ConsoleApplication1
{
  class Program
   {
      static void Main()
      {
         int i = int.Parse(Console.ReadLine());
         byte j;
         try
         {
            if (i > 255)
               // Генерируем исключение
               throw new OverflowException();
            else
               j = (byte)i;
         }
         catch (OverflowException)
         {
            Console.WriteLine("Возникло переполнение");
         }

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

Класс Exception

Класс **Exception** содержит несколько полезных свойств, с помощью которых можно получить информацию об исключении:

HelpLink	URL-файла справки с описанием ошибки
Message	Текстовое описание ошибки
Source	Имя объекта или приложения, которое сгенерировало ошибку
StackTrace	Последовательность вызовов, которые привели к возникновению ошибки. Свойство доступно только для чтения
InnerException	Содержит ссылку на исключение, послужившее причиной генерации текущего исключения
TargetSite	Метод, выбросивший исключение

Абстрактные классы

У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса просто непригоден для использования. Абстрактный метод автоматически становится виртуальным и не требует указания модификатора *virtual*. Для определения абстрактного метода служит приведенная ниже общая форма:

```
abstract [тип] [имя] (список_параметров);
```

У абстрактного метода отсутствует тело. Модификатор *abstract* может применяться только в методах экземпляра, но не в статических методах (*static*). Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением *class* указывается модификатор *abstract*. Попытка создать объект абстрактного класса с помощью оператора *new* приведет к ошибке во время компиляции!

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как *abstract*. Таким образом, атрибут *abstract* наследуется до тех пор, пока не будет достигнута полная реализация класса.

В абстрактные классы вполне допускается включать конкретные методы, которые могут быть использованы в своем исходном виде в производном классе. А переопределению в производных классах подлежат только те методы, которые объявлены как *abstract*.

Бесплодные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов. Если производный класс от абстрактного класса переопределяет не все методы, он также должен быть объявлен как абстрактный. В случае параметров абстрактного класса, его фактическим параметром может стать любой из его производных классов.

Интерфейсы

Интерфейс (*interface*) представляет собой не более чем просто именованный набор абстрактных членов. Абстрактные методы являются чистым протоколом, поскольку не имеют никакой стандартной реализации. Конкретные члены, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. Это действительно так. Интерфейс выражает поведение, которое данный класс или структура может избрать для поддержки. Более того, каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.

В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс - множество методов.

Интерфейсы объявляются с помощью ключевого слова *interface*. Ниже приведена упрощенная форма объявления интерфейса:

```
interface имя{
```

```
    возвращаемый_тип имя_метода_1 (список_параметров);
    возвращаемый_тип имя_метода_2 (список_параметров);
    // ...
    возвращаемый_тип имя_метода_N (список_параметров);
}
```

где имя - это конкретное имя интерфейса. В объявлении методов интерфейса используются только их возвращаемый_тип и сигнатура.

Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как static.

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе:

```
class имя_класса : имя_интерфейса {
    // тело класса
}
```

где *имя_интерфейса* - это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как *public*. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой. Кроме того, возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

Отличия интерфейса от абстрактного класса:

– элементы интерфейса по умолчанию имеют спецификатор доступа public и не иметь спецификаторов, заданных явным образом;

– интерфейс не может содержать полей и обычных методов – все элементы интерфейса должны быть абстрактными;

– класс, в списке предков которого задается интерфейс, должен определять все элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае пройти класс также будет абстрактным);

– класс может иметь в списке предков несколько интерфейсов.

В библиотеке .NET определено большое количество стандартных интерфейсов, которые описывают поведение объектов разных классов.

Работа с объектами через интерфейсы

При работе с интерфейсами бывает полезно убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции *is*, которая определяет совместимость текущего типа объекта с типом, заданным справа от нее.

Допустим, оформили какие-то действия с объектами в виде метода с параметром типа *object*. Прежде чем использовать этот параметр внутри метода для обращения к методам, созданным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно.

```
static void Act (object A)
{
    if ( A is IAction)  …..
```

```
}
```

Недостаток использования *is* – преобразование фактически выполняется дважды: при проверке и при преобразовании.

Более эффективной является операция *as*, которая выполняет преобразование к заданному типу, и если это невозможно, формирует результат *null*.

Обе операции применяются как в интерфейсах, так и в классах.

Интерфейсы и наследование

Интерфейс может иметь сколько угодно интерфейсов-предков, в том числе и ни одного. Он наследует все элементы всех своих предков, начиная с самого верхнего уровня. Нельзя использовать интерфейс, описанный со спецификатором *private* или *internal*, в качестве базового для открытого интерфейса.

В интерфейсе-потомке можно использовать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово *new*.

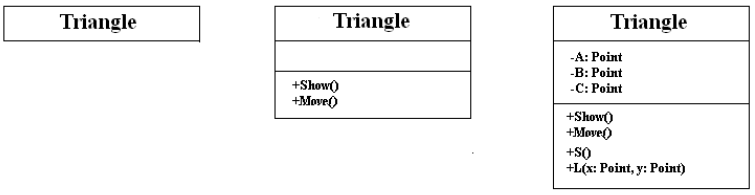
Отношения между классами. Диаграммы классов на языке UML

Язык унифицированного моделирования UML (Unified Modeling Language) является визуальным средством представления моделей программ, ставшим уже стандартом. Под моделями программ понимается их графическое представление в виде различных диаграмм, отражающих связи между объектами в программном коде. Одной из основных диаграмм UML является диаграмма классов, которая чрезвычайна удобна для сопоставления различных вариантов проектных решений.. Кроме того, она используется для описания шаблонов проектирования, которые в сочетании с объектно-ориентированной парадигмой программирования лежат в основе современного подхода к разработке программного обеспечения.

На диаграмме класс изображается в виде прямоугольника, состоящего из трех частей, расположенных вертикально. В верхней части указывается имя класса. В средней части приводится список полей, называемых в диаграмме UML атрибутами. Возможно, указание типов и атрибутов полей после двоеточия. Список методов (операций) приводится в нижней части, возможно, с указанием возвращаемого значения.

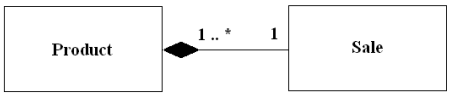
Имена абстрактных классов и операций принято обозначать курсивом. Перед именами полей и методов указывается спецификатор доступа с помощью одного из трех символов: + для public, - для private, # для protected. Для статических элементов класса после спецификатора доступа записывается символ $.

Во второй и третьих частях могут указываться не все элементы класса, а только представляющие интерес на данном уровне абстракции. Эти части могут быть совсем пусты, и в этом случае их можно не указывать.



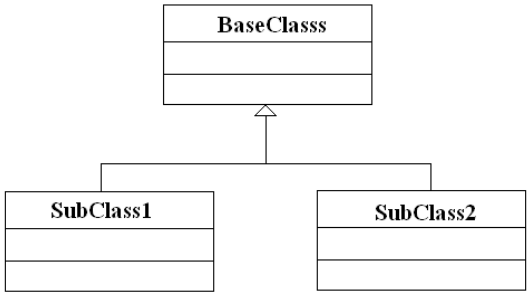
Существуют следующие отношения между классами: ассоциация, наследование, агрегация, зависимость.

Если два класса взаимодействуют друг с другом концептуально, то их взаимодействие называется ассоциацией. На диаграмме ассоциация показывается соединительной линией, над которой может быть указана кратность, т.е сколько объектов данного класса может быть связано с одним объектом другого класса. В случае произвольного количества указывается символ звездочка.

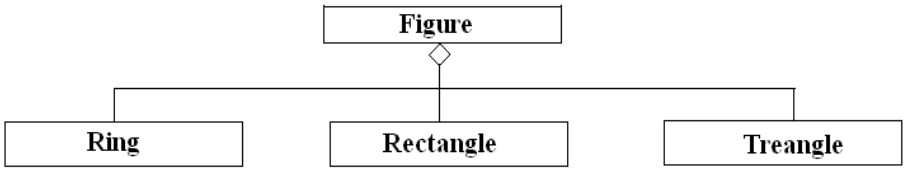


Ассоциация выявляется на ранней стадии проектирования, и в дальнейшем, как правило, конкретизируется.

Наследование на диаграмме классов показывается линией со стрелкой в виде незакрашенного треугольника, которая указывает на базовый класс. Допускается объединение несколько стрелок в одну для разгрузки диаграммы.

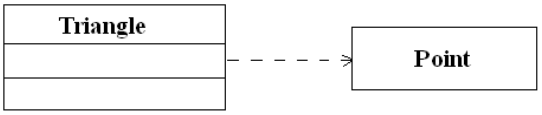


Отношение «агрегация» подразумевает включение в качестве составной части объектов другого класса (отношение целое/часть). На диаграмме такая связь обозначается стрелкой в виде незакрашенного ромба. При этом стрелка указывает на «целое». В программе для реализации нестрогой агрегации «часть» включается в «целое» по ссылке. Такой компонент может динамически появляться и исчезать.



Строгая агрегация имеет название – композиция. Она означает, что компонент не может исчезнуть, пока объект «целое» существует. Композиция обозначается линией со стрелкой в виде закрашенного ромба.

Отношение зависимости (использования) между классами показывает, что один класс пользуется некоторыми услугами другого класса. На диаграмме она обозначается пунктирной линией со стрелкой.



Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов. Интерфейс *IComparable* задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку. Реализация интерфейсов *IEnumerable* и *IEnuerator* дает возможность просматривать содержимое объекта с помощью *foreach*. Стандартные библиотеки поддерживаются многими стандартными классами. Можно создавать собственные классы, поддерживающие интерфейсы.

Интерфейс IComparable

Если требуется отсортировать коллекцию, состоящую из объектов определяемого пользо-

вателем класса, при условии, что они не сохраняются в коллекции класса *SortedList*, где элементы располагаются в отсортированном порядке, то в такой коллекции должен быть известен способ сортировки содержащихся в ней объектов. С этой целью можно, в частности, реализовать интерфейс *IComparable* для объектов сохраняемого типа. Интерфейс *IComparable* доступен в двух формах: обобщенной и необобщенной. Несмотря на сходство применения обеих форм данного интерфейса, между ними имеются некоторые, хотя и небольшие, отличия.

Если требуется отсортировать объекты, хранящиеся в необобщенной коллекции, то для этой цели придется реализовать необобщенный вариант интерфейса *IComparable*. В этом варианте данного интерфейса определяется только один метод, *CompareTo()*, который определяет порядок выполнения самого сравнения. Ниже приведена общая форма объявления метода *CompareTo()*:

```
int CompareTo(object obj);
```

В методе *CompareTo()* вызывающий объект сравнивается с объектом *obj*. Для сортировки объектов по нарастающей конкретная реализация данного метода должна возвращать нулевое значение, если значения сравниваемых объектов равны; положительное — если значение вызывающего объекта больше, чем у объекта *obj*; и отрицательное — если значение вызывающего объекта меньше, чем у объекта *obj*. А для сортировки по убывающей можно обратить результат сравнения объектов. Если же тип объекта *obj* не подходит для сравнения с вызывающим объектом, то в методе *CompareTo()* может быть сгенерировано исключение *ArgumentException*.

Если требуется отсортировать объекты, хранящиеся в обобщенной коллекции, то для этой цели придется реализовать обобщенный вариант интерфейса *IComparable<T>*. В этом варианте интерфейса *IComparable* определяется приведенная ниже обобщенная форма метода *CompareTo()*:

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
   class AutoShop : IComparable<AutoShop>
   {
      public string CarName { set; get; }
      public int MaxSpeed { get; set; }
      public double Cost { get; set; }
      public byte Discount { get; set; }
      public int ID { get; set; }

      public AutoShop() { }
      public AutoShop(string CarName, int MaxSpeed, double Cost, byte Discount, int ID)
      {
         this.CarName = CarName;
         this.MaxSpeed = MaxSpeed;
         this.Cost = Cost;
         this.Discount = Discount;
         this.ID = ID;
      }

      // Реализуем интерфейс IComparable<T>
      public int CompareTo(AutoShop obj)
      {
         if (this.Cost > obj.Cost)
            return 1;
         if (this.Cost < obj.Cost)
            return -1;
         else
```

```
            return 0;
        }

        public override string ToString()
        {
            return String.Format("{4}\tМарка: {0}\tМакс. скорость: {1}\tЦена: {2:C}\tСкидка:
{3}%",
            this.CarName,this.MaxSpeed,this.Cost,this.Discount,this.ID);
        }
    }

    class Program
    {
        static void Main()
        {
            List<AutoShop> dic = new List<AutoShop>();

            // Создадим множество автомобилей
            dic.Add(new AutoShop("Toyota Corolla", 180, 300000, 5, 1));
            dic.Add(new AutoShop("VAZ 2114i", 160, 220000, 0, 2));
            dic.Add(new AutoShop("Daewoo Nexia", 140, 260000, 5, 3));
            dic.Add(new AutoShop("Honda Torneo", 220, 400000, 7, 4));
            dic.Add(new AutoShop("Audi R8 Best", 360, 4200000, 3, 5));

            Console.WriteLine("Исходный каталог автомобилей: \n");
            foreach (AutoShop a in dic)
                Console.WriteLine(a);

            Console.WriteLine("\nТеперь автомобили отсортированны по стоимости: \n");
            dic.Sort();//становится возможна сортировка
            foreach (AutoShop a in dic)
                Console.WriteLine(a);

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

Если несколько объектов имеют одинаковые значения критерия сортировки, их порядок не изменится.

Если класс реализует интерфейс *IComparable*, его экземпляры можно сравнивать между собой по принципу больше или меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. Перегрузка операций обычно выполняется путем детегирования, то есть обращения к переопределенным методам *CompareTo* и *Equals*.

Пространство имен System.Collection

В классе *ArrayList* поддерживаются динамические массивы, расширяющиеся и сокращающиеся по мере необходимости. В языке C# стандартные массивы имеют фиксированную длину, которая не может изменяться во время выполнения программы. Это означает, что количество элементов в массиве нужно знать заранее. Но иногда требуемая конкретная длина массива остается неизвестной до самого момента выполнения программы. Именно для таких ситуаций и предназначен класс *ArrayList*. В классе *ArrayList* определяется массив переменной длины, который состоит

из ссылок на объекты и может динамически увеличивать и уменьшать свой размер.

В классе *ArrayList* определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Коллекцию класса *ArrayList* можно отсортировать, вызвав метод *Sort()*. В этом случае поиск в отсортированной коллекции с помощью метода *BinarySearch()* становится еще более эффективным. Содержимое коллекции типа *ArrayList* можно также обратить, вызвав метод *Reverse()*. Некоторые из наиболее часто используемых методов класса *ArrayList* перечислены ниже:

AddRange() - добавляет элементы из коллекции в конец вызывающей коллекции типа ArrayList.

BinarySearch() - выполняет поиск в вызывающей коллекции значения. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован.

CopyTo() - копирует содержимое вызывающей коллекции в массив, который должен быть одномерным и совместимым по типу с элементами коллекции.

FixedSize() - заключает коллекцию в оболочку типа ArrayList с фиксированным размером и возвращает результат. Возвращает часть вызывающей коллекции типа ArrayList. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу, и включает количество элементов, определяемое параметром count. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект.

IndexOf() - возвращает индекс первого вхождения объекта в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1.

InsertRange() - вставляет элементы коллекции в вызывающую коллекцию, начиная с элемента, указываемого по индексу.

Readonly() - заключает коллекцию в оболочку типа ArrayList, доступную только для чтения, и возвращает результат.

RemoveRange() - удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу index, и включая количество элементов, определяемое параметром count.

Sort() - сортирует вызывающую коллекцию по нарастающей.

В классе *ArrayList* поддерживается также ряд методов, оперирующих элементами коллекции в заданных пределах. Так, в одну коллекцию типа *ArrayList* можно вставить другую коллекцию, вызвав метод *InsertRange()*. Для удаления из коллекции элементов в заданных пределах достаточно вызвать метод *RemoveRange()*. А для перезаписи элементов коллекции типа *ArrayList* в заданных пределах элементами из другой коллекции служит метод *SetRange()*. И наконец, элементы коллекции можно сортировать или искать в заданных пределах, а не во всей коллекции.

Пример:

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
   class MyCollection
   {
      public static ArrayList NewCollection(int i)
      {
         Random ran = new Random();
         ArrayList arr = new ArrayList();

         for (int j = 0; j < i; j++)
            arr.Add(ran.Next(1, 50));
         return arr;
      }
```

```
        public static void RemoveElementMyCollection(int i, int j, ref ArrayList arr)
        {
            arr.RemoveRange(i, j);
        }

        public static void AddElementInMyCollection(int i, ref ArrayList arr)
        {
            Random ran = new Random();
            for (int j = 0; j < i; j++)
                arr.Add(ran.Next(1, 50));
        }

        public static void WriteMyCollection(ArrayList arr)
        {
            foreach (int a in arr)
                Console.Write("{0}\t", a);
            Console.WriteLine("\n");
        }
    }

    class Program
    {
        static void Main()
        {
            // Создадим новую коллекцию чисел длиной 8
            ArrayList Coll = MyCollection.NewCollection(8);
            Console.WriteLine("Исходная коллекция чисел: ");
            MyCollection.WriteMyCollection(Coll);

            // Удалим пару элементов
            MyCollection.RemoveElementMyCollection(5, 2, ref Coll);
            Console.WriteLine("Коллекция после удаления предпоследних двух элементов: ");
            MyCollection.WriteMyCollection(Coll);

            // Добавим еще несколько элементов
            MyCollection.AddElementInMyCollection(10, ref Coll);
            Console.WriteLine("Добавили 10 элементов: ");
            MyCollection.WriteMyCollection(Coll);

            // Отсортируем теперь коллекцию
            Coll.Sort();
            Console.WriteLine("Отсортированная коллекция: ");
            MyCollection.WriteMyCollection(Coll);

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

Паттерны проектирования

Па́ттерн (англ. 'pattern – образец, шаблон, система) – заимствованное слово. Смысл термина

«паттерн» всегда у́же чем просто «образец», и варьируется в зависимости от области знаний, в которой используется. Слово pattern, в зависимости от контекста, имеет широкий диапазон значений.

Паттерн (информатика) – эффективный способ решения характерных задач проектирования, в частности, задач проектирования компьютерных программ.

Синоним термина – шаблон проектирования.

Обычно шаблон не является законченным образцом, который может быть напрямую преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.

Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

Первые шаблоны проектирования для языка программирования Smalltalk представили в 1987 г. Кэнт Бэк и Вард Каннингем. Но это было только началом, настоящее признание в мире программирования они получили после публикации книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования», написанной программистами Э. Гаммом, Р. Хелмом, Р. Джонсоном и Дж. Влиссидесом.

В ней были представлены 23 шаблона, ставших сейчас основными. Данная работа дала толчок к изучению паттернов программистами. Издание «банды четырех» (так в шутку прозвали авторов книги) до сих пор остается одним из ИТ-бестселлеров, и его постоянно публикуют.

В целом паттерны представляют собой некую архитектурную конструкцию, помогающую описать и решить определенную общую задачу проектирования. Они приобрели такую популярность потому, что разработка ПО ко времени их формализации уже была достаточно развита. Многие понимали, что не стоит изобретать велосипед, а использование паттернов часто бывает полезным как отдельному разработчику, так и целой команде.

Впрочем, применение шаблонов проектирования связано и с определенными проблемами. В частности, распространено мнение, что только специалист, обладающий достаточно высокой квалификацией и понимающий, какие из паттернов ему нужны, сумеет правильно использовать их в своих программах. Кроме того, зачастую некоторые разработчики, изучившие лишь несколько шаблонов проектирования, начинают употреблять их повсюду, даже там, где они не слишком хорошо справляются с задачей и усложняют создаваемое ПО.

«Банда четырех» разделила паттерны проектирования на три основные группы:

- порождающие — призванные создавать объекты;
- структурные — меняющие структуру взаимодействия между классами;
- поведенческие — отвечающие за поведение объектов.

Кстати, в наши дни несложно выделить и другие группы и даже антипаттерны, подсказывающие, как не надо разрабатывать ПО.

Abstract Factory – это один из самых известных порождающих шаблонов проектирования. Он позволяет разработчику создать интерфейс для объектов, каким-либо образом связанных между собой. Причем не требуется указывать конкретные классы, поскольку работать с каждым из них можно будет через этот интерфейс. С помощью такой фабрики удастся создавать группы объектов, реализующих общее поведение. Преимущество данного паттерна заключается в том, что он изолирует конкретные классы, благодаря чему легко заменять семейства продуктов. А к его недостаткам следует отнести то, что при расширении возможностей фабрики путем добавления нового типа продуктов придется редактировать все конкретные реализации Abstract Factory, а это порой бывает недопустимо, например, если уже создано 100 конкретных фабрик.

Рассмотрим, как на практике можно использовать этот паттерн. Сначала создадим абстрактную фабрику CarFactory, содержащую семейство из двух объектов — автомобиля и двигателя для него.

```
abstract class CarFactory
{
  public abstract AbstractCar CreateCar();
  public abstract AbstractEngine CreateEngine();
```

```
 }
```
Теперь реализуем первую конкретную фабрику, создающую класс, описывающий автомобиль BMW и двигатель для него:
```
class BMWFactory : CarFactory
 {
  public override AbstractCar CreateCar()
  {
   return new BMWCar();
  }
public override AbstractEngine CreateEngine()
  {
   return new BMWEngine();
  }
 }
```
Сделаем то же самое для автомобиля марки Audi, чтобы у нас возникла вторая конкретная фабрика:
```
class AudiFactory : CarFactory
 {
  public override AbstractCar CreateCar()
  {
   return new AudiCar();
  }
public override AbstractEngine CreateEngine()
  {
   return new AudiEngine();
  }
 }
```
Теперь опишем абстрактный класс для наших автомобилей. В данном случае у них будет один метод, позволяющий узнать максимальную скорость машины. С его помощью мы обратимся и ко второму объекту — двигателю:

```
abstract class AbstractCar
{
 public abstract void MaxSpeed(AbstractEngine engine);
}
```

Все двигатели, в свою очередь, будут содержать один параметр — максимальную скорость. Эта простая общедоступная переменная позволит сократить объем программы в данном примере:

```
abstract class AbstractEngine
 {
   public int max_speed;
 }
```
Реализуем класс для автомобиля BMW:
```
class BMWCar : AbstractCar
{
public override void MaxSpeed(AbstractEngine engine)
{
Console.WriteLine(«Максимальная скорость: « + engine.max_speed.ToString());
}
 }
```

А затем определяем параметры его двигателя:

```
class BMWEngine : AbstractEngine
 {
  public BMWEngine()
 {
   max_speed = 200;
 }
 }
```

Проделаем то же самое для класса, описывающего автомобиль Audi:

```
class AudiCar : AbstractCar
  {
public override void MaxSpeed(AbstractEngine engine)
{
Console.WriteLine(«Максимальная скорость: « + engine.max_speed.ToString());
}
}
```

Задаем двигатель для него:

```
class AudiEngine : AbstractEngine
  {
  public AudiEngine()
  {
   max_speed = 180;
  }
  }
```

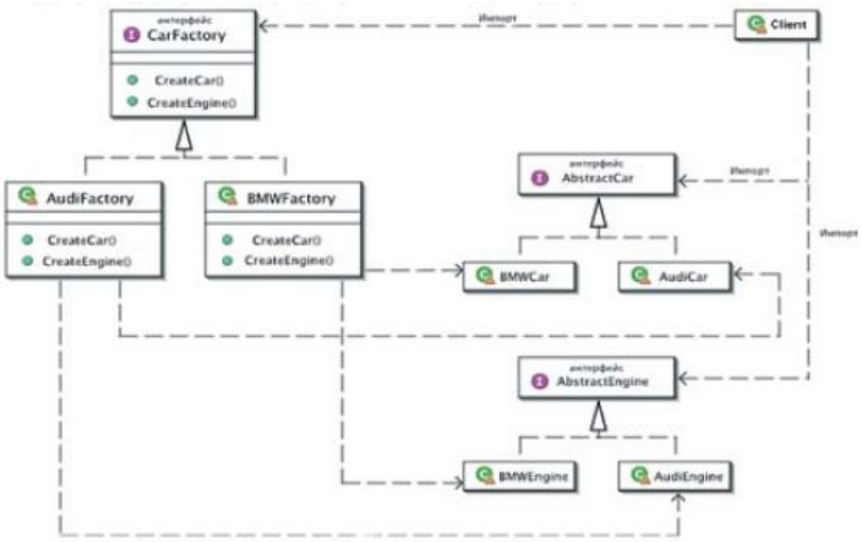
Теперь мы создадим класс Client, где покажем, как осуществляется работа с абстрактной фабрикой. В конструктор такого класса будут передаваться все конкретные фабрики, которые и начнут создавать объекты автомобиль и двигатель. Следовательно, в конструктор класса Client допустимо передать любую конкретную фабрику, работающую с любыми марками автомобилей. А метод Run позволит узнать максимальную скорость конкретной машины.

```
 class Client
 {
   private AbstractCar abstractCar;
private AbstractEngine abstractEngine;
public Client(CarFactory car_factory)
   {
     abstractCar = car_factory.CreateCar();
     abstractEngine = car_factory.CreateEngine ();
   }
public void Run()
   {
      abstractCar.MaxSpeed(abstractEngine);
    }
  }
```

Ниже показано, как вызвать метод Run с различными параметрами:

```
   public static void Main()
   {
     // Абстрактная фабрика № 1
CarFactory bmw_car = new BMWFactory ();
     Client c1 = new Client(bmw_car);
```

```
    c1.Run();
// Абстрактная фабрика № 2
CarFactory audi_factory = new AudiFactory();
Client c2 = new Client(audi_factory);
c2.Run();
    Console.Read();
   }
```



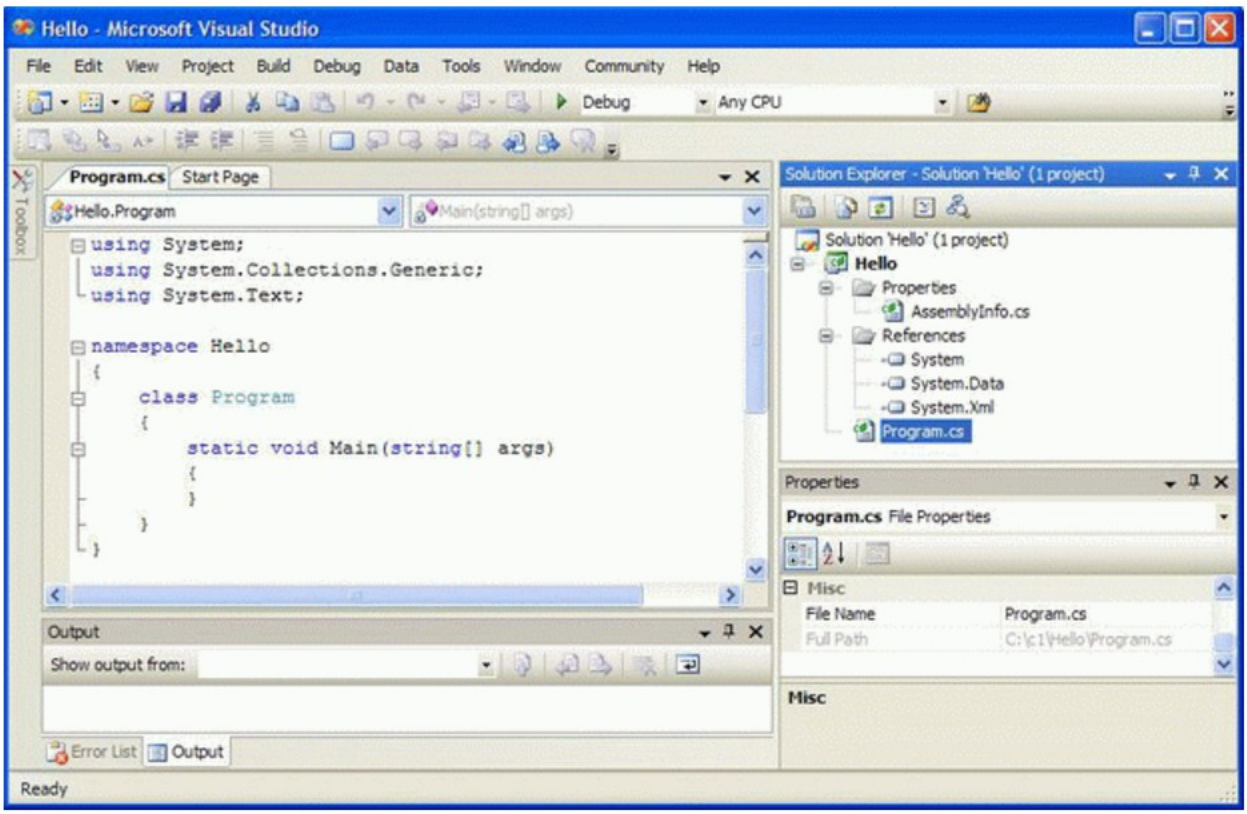
МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

Лабораторная работа 1

Создание проекта на языке Visual C# в среде Visual Studio .NET
Типы данных в C#. Операторы ввода-вывода. Классы в C#

Шаги создания проекта:

1. Запустите среду Visual Studio 2008 (или Visual Studio 2005).
2. В меню **Файл** выберите пункт **Создать проект.** (File→New→Project) Появится диалоговое окно **Создание проекта**.
3. В левой части диалогового окна **Типы проектов Templates** выберите пункт **Visual C# Projects** , а в правой – пункт Console Application.
4. В поле **Имя** (Name) введите имя нового приложения, а в поле Location – место его сохранения на диске.
5. После щелчка по кнопке ОК среда создаст решение и проект с указанным именем.



6. В верхней части экрана располагается главное меню и панели инструментов.
7. В правой нижней части экрана располагается окно свойств **Properties**. Если оно закрыто, то его можно включить командой **View - Properties**. В этом окне отображаются важнейшие характеристики выделенного элемента.
8. Основное пространство экрана занимает окно редактора, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист будет добавлять нужный код. При этом зарезервированные слова отображаются синим цветом, комментарии – зеленым, основной текст – черным.
9. Текст структурирован. Щелкнув на знак минус, мы скроем блок кода, щелкнув на знаке плюс – откроем.
10. Открыв папку, содержащую проект, и рассмотрим ее структуру.

На данном этапе особый интерес для нас будут представлять следующие файлы:

Hello.sln – основной файл, отвечающий за весь проект. Если необходимо открыть проект для редактирования, то нужно выбрать именно этот файл. Остальные файлы откроются автоматически.

Program.cs – файл, в котором содержится исходный код - код, написанный на языке C#. Именно с этим файлом мы и будем непосредственно работать.

Hello.exe – файл, в котором содержатся сгенерированный IL-код и метаданные проекта. Другими словами, этот файл и есть готовое приложение, которое может выполняться на любом компьютере, на котором установлена платформа .Net.

C# - объектно-ориентированный язык, поэтому написанная на нем программа будет представлять собой совокупность взаимодействующих между собой классов. Автоматически был создан класс с именем Program (в других версиях среды может создаваться класс с именем Class1).

Варианты

1. Создать класс *комплексное число,* содержащий:
поля: вещественная и мнимая часть числа;
методы: ввод полей с клавиатуры, нахождение значения аргумента и модуля, перевод числа в тригонометрическую форму, вывод числа на экран в общепринятом виде.
2. Создать класс *комплексное число* содержащий:
поля: модуль и аргумент;
методы: ввод полей с клавиатуры, нахождения вещественной и мнимой частей; вывода числа на экран в общепринятом виде.
3. Создать класс *вектор* на плоскости, содержащий:
поля: координаты начала и окончания вектора;
методы: ввод полей с клавиатуры, вывод координат на экран, длина вектора, угол между вектором и осью OX.
4. Создать класс *вектор* в пространстве содержащий:
поля: координаты начала и окончания вектора;
методы: ввод полей с клавиатуры, длина вектора, нахождение координат вектора, вывод координат вектора.
5. Создать класс *вектор* в пространстве содержащий:
поля: координаты начала и окончания вектора;
методы: ввод полей с клавиатуры, вывод координат вектора на экран, нахождение угла, образованного вектором с осью OX, длина вектора
6. Организовать класс *треугольник,* содержащий:
поля: координатам вершин;
методы: ввод полей с клавиатуры, нахождение длин сторон, периметр треугольника, длина высоты на большую сторону, вывода значений полей на экран.
7. Организовать класс *треугольник,* содержащий:
поля: длины сторон;
методы: ввод полей с клавиатуры, периметра, площадь (по формуле Герона), вывод на экран значений полей.
8. Организовать класс *параллелограмм*, содержащий:
поля: длины сторон и меньшим углом между ними,
методы: ввод полей с клавиатуры, нахождение периметра, площадь параллелограмма, длина его диагоналей.
9. Организовать класс *прямоугольный треугольник,* содержащий:
поля: длины катетов;
методы: ввод полей с клавиатуры, гипотенуза, периметра, вывод на экран значений полей.

10. Организовать класс *прямоугольный треугольник,* содержащий:

поля: длины катетов;

методы: площадь, величина углов, вывод на экран значений полей.

11. Организовать класс правильный многоугольник, содержащий:

поля: количество сторон, длина стороны,

методы: ввод полей с клавиатуры, периметр, величина угла, вывод полей на экран.

12. Организовать класс *круг,* содержащий:

поля: координаты центра, радиус,

методы: ввод полей с клавиатуры, длины окружности, площади, вывода на экран полей.

13. Описать класс *прямоугольник*, содержащий:

поля: длины сторон,

методы: периметр, площадь, длина диагонали, вывод на экран полей класса.

14. Описать класс *ромб*, содержащий:

поля: длины диагоналей, величина меньшего угла,

методы: ввод полей с клавиатуры, длина стороны, величина большего угла, вывода на экран всех полей.

15. Организовать класс *дробь*, содержащий:

поля: числитель и знаменатель,

методы: ввод полей с клавиатуры, вывод дроби в общепринятом виде, выделение целой части, вывод дроби с целой частью.

16. Организовать класс *правильная дробь,* содержащий:

поля: целая часть, числитель и знаменатель,

методы: ввод полей с клавиатуры, вывода дроби в общепринятом виде, приведение дроби к несократимому виду.

17. Описать класс *число*, содержащий:

поля: значение числа в десятичной форме, значение числа в восьмеричной форме,

методы: ввод поля десятичное число с клавиатуры, перевод числа из десятичной формы в восьмеричную, вывод значений полей на экран,.

18. Описать класс *число*, содержащий

поля: значение числа в десятичной форме, значение числа в двоичной форме, методы: ввод значения поля десятичное число с клавиатуры, перевода числа из десятичной формы в двоичную, вывода значений полей.

19. Описать класс *точка*, содержащий:

поля: координата точки в двумерном пространстве;

методы: ввод значения поля с клавиатуры, расстояние от точки до начала координат, вывод значений полей на экран.

20. Описать класс *точка*, содержащий

поля: координата точки в трехмерном пространстве;

методы: ввод значения поля с клавиатуры, расстояние от точки до начала координат, вывод значений полей на экран.

Лабораторная работа 2
Элементы класса: конструкторы, свойства

Конструктор – это метод класса, предназначенный для инициализации объекта при его создании.

Инициализация – это задание начальных параметров объектов/переменных при их создании.

Особенностью конструктора, как метода, является то, что его имя всегда совпадает с име-

нем класса, в котором он объявляется. При этом, при объявлении конструктора, не нужно указывать возвращаемый тип, даже ключевое слово *void*. Конструктор следует объявлять как *public*, иначе объект нельзя будет создать (хотя иногда в этом также есть смысл). В классе возможно объявлять множество конструкторов, главное чтобы они отличались между собой сигнатурами - набором аргументов.

Свойство в C# – это член класса, который предоставляет удобный механизм доступа к полю класса (чтение поля и запись). Свойство представляет собой что-то среднее между полем и методом класса. При использовании свойства, мы обращаемся к нему, как к полю класса, но на самом деле компилятор преобразовывает это обращение к вызову соответствующего неявного метода. Такой метод называется аксессор (*accessor*). Существует два таких метода: *get* (для получения данных) и *set*(для записи). Объявление простого свойства имеет следующую структуру:

[модификатор доступа] [тип] [имя_свойства]

```
{
  get
  {
    // тело аксессора для чтения из поля
  }

  set
  {
    // тело аксессора для записи в поле
  }
}
```

Пример использования свойств. Имеется класс *Студент*, и в нем есть закрытое поле курс, которое не может быть ниже единицы и больше пяти. Для управления доступом к этому полю будет использовано свойство *Year*:

```
class Student
{
  private int year; //объявление закрытого поля

  public int Year //объявление свойства
  {
   get // аксессор чтения поля
   {
      return year;
   }
   set // аксессор записи в поле
   {
       if (value < 1)
         year = 1;
       else if (value > 5)
         year = 5;
       else year = value;
    }
  }
}
class Program
{
  static void Main(string[] args)
  {
    Student st1 = new Student();
    st1.Year = 0; // записываем в поле, используя аксессор  set
```

Console.WriteLine(st1.Year); *// читаем поле, используя аксессор get,*

Задание

Модифицируйте программу предыдущей лабораторной работы, дополнив все закрытые поля свойствами и объявив в классе три конструктора. Продемонстрируйте работу всех методов.

Лабораторная работа 3

Работа с массивами в языке C#

Создание массива

В языке C# массива относятся к ссылочным типам данных и располагаются в динамической памяти. Поэтому создание массива начинается с ее выделения.

Синтаксис объявления одномерного массива:

тип [ ] имя;

тип [ ] имя = new тип[количество элементов];

тип [ ] имя = new тип[количество элементов]{список значений};

тип [ ] имя = {список значений};

тип [ ] имя = new тип[ ]{список значений};

Например,

```
int [ ] x = new int[10];
string [ ] s = new string [50];
int n = 15;  int [ ] y = new int[n];
int [ ] A = new int[ ] {10, 20, 30, 40, 50};
int [ ] B = new int[3] {-1, 2, -3};
```

Элементы массива нумеруются с нуля. Для обращения к элементу после имени в квадратных скоках указывается его индекс (номер).

Массивы можно создавать любого типа, в том числе и типа класса. В этом случае получается массив оъектов.

Прямоугольные массивы имеют более одной размерности. Размерности указываются через запятую. Например,

```
int [ , ] x = new int[10, 5];
int [ , ] B = new int[2, 3] {{-1, 2, -3}, {0, 1, 3}};
```

В ступенчатых массивах количество элементов в разных строках может различаться. В памяти ступенчатый массив в виде нескольких внутренних массивов. Пример описания ступенчатого массива:

```
int [ ] [ ] x = new int[3] [ ];
x[0] = new int [5];
x[1] = new int [7];
x[2] = new int [4];
```

При обращении к ступенчатому массиву индекс каждой размерности указывается в отдельной паре квадратных скобок.

Количество элементов в массиве (*размерность*) задается при выделении памяти и не может быть изменена впоследствии. Она может задаваться выражением:

```
short n = ...;
string[] z = new string[2*n + 1];
```

Размерность не является частью типа массива.

Элементы массива нумеруются *с нуля*.

Для *обращения к элементу массива* после имени массива указывается номер элемента в квадратных скобках, например:

w[4] z[i]

С элементом массива можно делать все, что допустимо для переменных того же типа.

При работе с массивом автоматически выполняется *контроль выхода за его границы*: если значение индекса выходит за границы массива, генерируется исключение IndexOutOfRangeException.

Базовый класс Array

Массивы в C# построены на основе базового класса Array, который с одержит набор полезных свойств и методов.

Элемент	Вид	Описание
Length	свойство	Количество элементов по всем размерностям
Rank	свойство	Количество размерностей
BinarySearch	статический метод	Двоичный поиск в отсортированном массиве
Clear	статический метод	Присваивание элементам массива значений по умолчанию
Copy	статический метод	Копирование заданного диапазона элементов одного массива в другой
CopyTo	метод	Копирование всех элементов текущего одномерного массива в другой
GetValue	метод	Получение значения элемента массива
Index0f	статический метод	Поиск первого вхождения элемента в одномерный массив
LastIndex0f	статический метод	Поиск последнего вхождения элемента в одномерный массив
Reverse	статический метод	Изменение порядка следования элементов на обратный
SetValue	метод	Установка значения элемента массива
Sort	статический метод	Упорядочивание элементов одномерного массива

К статическим элементам обращаются через имя класса, а не его экземпляра.

В классе Class1 описан вспомогательный статический метод PrintArray, предназначенный для вывода значений массива на экран. При этом значение элемента массива нужно получат методом GetValue. Например,

```
public static void PrintArray (string header, Array a)
{
Console.WriteLine (header);
for ( int i = 0; i < a.Length; ++i )
        Console.WriteLine ( “\t” + a.GetValue(i) );
Console.WriteLine ();
}
```

Оператор foreach

Для перебора значений массива применяется оператор foreach.

Синтаксис оператора

foreach (тип имя in выражение) тело цикла

С помощью имени задается локальная переменная, принимающая все значения массива выражения по очереди. Например,

```
int [ ] x = {5, -3, 2, -7, 8,  4};
long s=0;
```

```
int k = 0;
foreach (int elem in x)
        if ( elem < 0 )   { s += elem; ++num; }
```

Класс Random

Класс Random определен в пространстве имен System и позволяет генерировать случайным образом данные.

Для получения последовательности положительных чисел требуется сначала создать экземпляр класса Random с помощью одного из его конструкторов. Например,

```
Random a = new Random( ); //конструктор по умолчанию
Random b = new Random( 1 ); //конструктор с параметром
```

Во втором случае обеспечивается возможность получения одинаковых последовательностей чисел.

Основные методы класса Random

Имя	Действие
Next( )	Возвращает целое положительное число во всем положительном диапазоне типа int
Next(макс)	Возвращает целое положительное число в диапазоне [0, макс]
Next(мин, макс)	Возвращает целое положительное число в диапазоне [мин, макс]
NextBytes(массив)	Возвращает массив чисел в диапазоне [0, 255]
NextDouble( )	Возвращает вещественное положительное число в диапазоне [0, 1]

Пример

```
Random b = new Random( 1 );
for ( int i = 0; i < 10;  ++i )
Console.Write( “  ” + b.Next (1000) );
```

Массив как параметр функции

Так как имя массива фактически является ссылкой, то он передается в метод по ссылке и, следовательно, все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива, являющимся фактическим параметром.

Пример

```
class Program
{
 static void Print(int n, int[ ] a) //n - размерность массива, a - ссылка на массив
 {
  for (int i = 0; i < n; i++) Console.Write("{0} ", a[i]);
  Console.WriteLine();
 }

 static void Change(int n, int[ ] a)
 {
  for (int i = 0; i < n; i++)
  if (a[i] > 0) a[i] = 0; // изменяются элементы массива
 }

 static void Main()
 {
  int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };
```

```
    Print(10, myArray);
    Change(10, myArray);
    Print(10, myArray);
  }
}
```

Задание

Создать программу обработки элементов двумерного массива. В программе задание элементов массива предусмотреть двумя способами (на выбор пользователя): с клавиатуры и случайным образом.

Варианты

1. Подсчитать количество локальных минимумов матрицы размерности 6 на 6. Локальным минимумом называется элемент, значение которого меньше всех его соседей.
2. Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (по выбору пользователя). Значение n вводить с клавиатуры.
3. Определить количество одинаковых элементов в каждой строке матрицы размерности 8 на 10.
4. Определить номера строк матрицы размерности 10 на 10, элементы которых полностью совпадают с элементами столбца с таким же номером.
5. Осуществить циклический сдвиг элементов ступенчатого массива на n элементов вправо. Значение n вводит с клавиатуры.
6. Переписать элементы ступенчатого массива с 10 строками в обратном порядке в каждой строке.
7. Последний элемент ступенчатого массива с 10 строками заменить суммой элементов, расположенных до него.
8. Изменить содержимое матрицы размерности 10 на 10, удалив из нее отрицательные элементы со сдвигом вправо.
9. Изменить содержимое матрицы 10 на 10, поменяв ее элементы симметрично расположенные относительно главной диагонали. Элементы главной диагонали записать в обратном порядке.
10. Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов слева или вверх (по выбору пользователя). Значение n вводит с клавиатуры.
11. Подсчитать количество локальных максимумов матрицы размерности 6 на 6. Локальным максимумом называется элемент, значение которого больше всех его соседей.
12. Последний элемент ступенчатого массива с 10 строками заменить минимальным элементом из элементов, расположенных до него.
13. Изменить содержимое квадратной матрицы размерности 8 на 8, поменяв элементы главной диагонали с максимальными элементами соответствующих строк.
14. Изменить содержимое квадратной матрицы размерности 8 на 8, поменяв элементы главной диагонали с минимальными элементами соответствующих столбцов.
15. Изменить содержимое квадратной матрицы размерности 8 на 8, поменяв элементы главной диагонали с элементами побочной диагонали.
16. В ступенчатом массиве определить количество строк, среднее арифметическое элементов которых меньше заданной величины.
17. Уплотнить ступенчатый массив, удаляя из него все элементы, равные нулю. При удалении элемента производить сдвиг элементов влево.
18. Характеристикой строки матрицы называют сумму ее отрицательных четных элементов. Определить количество строк матрицы 6 на 8, характеристика которых меньше -10.
19. Матрица A имеет седловую точку A_{ij}, если A_{ij} является минимальным элементом в i-ой строке и максимальным в j-ом столбце. Определить местоположение всех седловых точек матрицы размерности 8 на 8.

20. Уплотнить заданную матрицу размерности 10 на 10, удалив из нее все строки и столбцы, заполненные нулями

Лабораторная работа 4

Тема Наследование. Перегрузка операций

Задание. Измените программу предыдущего задания, организовав на основе существующего класса производный класс. Продемонстрируйте работу всех методов базового и производного классов. Функцию вывода полей класса сделать виртуальной, реализовав ее тело в базовом и производных классах различными способами.

Описание классов расположите в отдельном файле, подключив его к основной программе.

Постройте UML-диаграмму классов для созданного приложения.

Варианты

1. Организовать производный класс, содержащий бинарные операции: сложения, вычитания, равно, не равно; унарные операции: инкремента, декремента комплексных чисел.
2. Организовать производный класс, содержащий бинарные операции: умножения, деления, равно, не равно; унарные операции: инкремента, декремента комплексных чисел.
3. Организовать производный класс, дополнительно содержащий операции сложения векторов и вычитания, а также операции инкремента и декремента и метод нахождения угла между вектором и осью OX.
4. Организовать производный класс, дополнительно содержащий операции сложения, произведение векторов, а также операции инкремента и декремента, неравенства и равенства векторов.
5. Организовать производный класс, дополнительно содержащий операции сложения, произведение векторов, а также операции инкремента и декремента и равенства векторов и метод нахождения угла между вектором и осью OX.
6. Описать производный класс *треугольная призма*, добавив в него поле «высота» и методы: нахождения объема призмы и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
7. Описать производный класс *пирамида*, добавив в него поле «высота» и методы: нахождения объема пирамиды и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
8. Описать производный класс *призма*, добавив в него поле «высота» и методы: нахождения объема призмы и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
9. Описать производный класс *призма*, добавив в него поле «высота» и методы: нахождения объема призмы и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
10. Описать производный класс *призма*, добавив в него поле «высота» и методы: нахождения объема призмы и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
11. Описать производный класс *пирамида*, добавив в него поле «высота» и метод нахождения площади поверхности пирамиды. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине площади).
12. Описать производный класс *цилиндр*, добавив в него поле «высота» и методы: нахождения объема цилиндра и площади поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
13. Описать производный класс *параллелепипед*, добавив в него поле «высота» и методы: нахождения объема параллелепипеда, и площади его поверхности. В классе реализовать операции равно, не равно, больше, меньше (сравнение проводить по величине объема).
14. Описать производный класс *призма*, добавив в него поле «высота» и методы: нахождения объема призмы и площади поверхности. В классе реализовать операции равно, не равно,

больше, меньше (сравнение проводить по величине объема).

15. Организовать производный класс, содержащий бинарные операции: деления и умножения дробей, равно, не равно, больше, меньше, унарную операцию инкремента, унарную операцию инкремента.

16. Описать производный класс, дополнительно содержащий бинарные операции сложения и вычитания чисел, операции сравнения (больше, меньше, равно, не равно), унарную операцию декремента.

17. Описать производный класс, дополнительно содержащий операции сложения и вычитания чисел, операции сравнения (больше, меньше, равно, не равно), инкремента и декремента.

18. Описать производный класс, дополнительно содержащий операции сложения и вычитания чисел, операции сравнения (больше, меньше, равно, не равно), инкремента и декремента.

19. Описать производный класс, дополнительно содержащий операции сравнения (больше, меньше, равно, не равно), инкремента и декремента. Операции «больше» и «меньше» реализовывать как сравнение расстояния от начала координат. Инкремент и декремент обеспечивает единичное приращение координат.

20. Описать производный класс, дополнительно содержащий операции сравнения (больше, меньше, равно, не равно), инкремента и декремента. Операции «больше» и «меньше» реализовывать как сравнение расстояния от начала координат. Инкремент и декремент обеспечивает единичное приращение координат.

Лабораторная работа 5

Событийное программирование в C#

Событийно-управляемое программирование

В основу ОС Windows положен принцип *событийного управления.* Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется *событием.*

Событие воспринимается Windows и преобразуется в *сообщение* запись, содержащую необходимую информацию о событии (например, какая клавиша была нажата, в каком месте экрана произошел щелчок мышью). Сообщения могут поступать не только от пользователя, но и от самой системы, а также от активного или других приложений. Определен достаточно широкий круг стандартных сообщений, образующий иерархию, кроме того, можно определят собственные сообщения.

Сообщения поступают в общую очередь, откуда распределяются по очередям приложений. Каждое приложение содержит *цикл обработки сообщений*, которое **выбирает сообщение из очереди и через операционную систему вызывает подпрограмму, предназначенную для его обработки. Таким образом, Windows**-приложение состоит из главной программы, содержащей цикл обработки сообщений, инициализацию и завершение приложения, и набора *обработчиков событий.*

Среда Visual Studio.NET содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу создание шаблонов и заготовок обработчиков событий.

Шаблон Windows-приложения

Для создания нового проекта (File ► New ► Project), нужно выбрать шаблон Windows Application.

Среда самостоятельно сформирует шаблон Windows-приложения.Среда создает не только заготовку формы, но и шаблон текста приложения. Перейти к нему можно, щелкнув в окне Solution Explorer (View ► Solution Explorer) правой кнопкой мыши на файле Forml.cs и выбрав в контекстном меню команду View Code.

Процесс создания Windows-приложения состоит из двух этапов:

1. Визуальное проектирование – задание внешнего облика приложения;
2. Определения поведения приложения путем кодирования процедур обработки событий.

Для визуального проектирования можно воспользоваться командой меню: View►Properties Windows. Самый простой способ размещения компонента - двойной щелчок на соот¬ветствующем значке палитры компонентов Toolbox (если ее не видно, можно воспользоваться командой меню View ► Toolbox), при этом компонент помещается на форму. Затем компонент можно переместить и изменить его размеры с помощью мыши. Можно также сделать один щелчок на палитре и еще один щелчок в том месте формы, где планируется разместить компонент.

Заготовка шаблона обработчика события формируется двойным щелчком на поле, расположенном справа от имени соответствующего события на вкладка Events.

Задание

Общая часть задания: написать Windows-приложение, заголовок главного окна которого содержит Ф. И. О., группу и номер варианта.

Варианты

Вариант 1. Создать меню с командами Input, Calc и Exit.

При выборе команды Input открывается диалоговое окно, содержащее:

- три поля типа TextBox для ввода длин трех сторон треугольника;
- группу из двух флажков (Периметр и Площадь) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода длин трех сторон треугольника;
- выбора режима с помощью флажков: подсчет периметра и/или площади треугольника.

При выборе команды Calc открывается диалоговое окно с результатами. При выборе команды Exit приложение завершается.

Вариант 2. Создать меню с командами Size, Color, Paint, Quit. Команда Paint недоступна. При выборе команды Quit приложение завершается.

При выборе команды Size открывается диалоговое окно, содержащее:

- два поля типа TextBox для ввода длин сторон прямоугольника;
- группу из трех флажков (Red, Green, Blue) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода длин сторон прямоугольника в пикселях в поля ввода;
- выбора его цвета с помощью флажков.

После задания параметров команда Paint становится доступной.

При выборе команды Paint в главном окне приложения выводится прямоугольник заданного размера и сочетания цветов или выдается сообщение, если заданные параметры превышают размер окна.

Вариант 3. Создать меню с командами Input, Work, Exit. При выборе команды Exit приложение завершает работу.

При выборе команды Input открывается диалоговое окно, содержащее:

- три поля ввода типа TextBox с метками Radius, Height, Density,
- группу из двух флажков (Volume, Mass) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода радиуса, высоты и плотности конуса;
- выбора режима с помощью флажков: подсчет объема и/или массы конуса.

При выборе команды Work открывается окно сообщений с результатами.

Вариант 4. Создать меню с командами Input, Calc, Draw, Exit. При выборе команды Exit приложение завершает работу.

При выборе команды Input открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой Radius;
- группу из двух флажков (Square, Length) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода радиуса окружности;
- выбора режима с помощью флажков: подсчет площади круга (Square) и/или длины окружности (Length).

При выборе команды Calc открывается окно сообщений с результатами.

При вы боре команды Draw в центре главного окна выводится круг введенного радиуса или выдается сообщение, что рисование невозможно (если диаметр превышает размеры рабочей области).

Вариант 5. Создать меню с командами Input, Calc, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Input открывается диалоговое окно, содержащее:

- три поля ввода типа TextBox с метками Number 1, Number 2, Number 3;
- группу из двух флажков (Summ, Least multiple) типа CheckBox,
- кнопку типа Button.

Обеспечить возможность ввода трех чисел и выбора режима вычислений с помощью флажков: подсчет суммы трех чисел (Summ) и/или наименьшего общего кратного двух первых чисел (Lmirit multiple).

При выборе команды Calc открывается диалоговое окно с результатами.

Вариант 6. Создать меню с командами Input, Calc, Quit. Команда Calc недоступна. При выборе команды Quit приложение завершается.

При выборе команды Input открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Number 1, Number 2;
- группу из трех флажков (Summa, Max divisor, Multiply) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода двух чисел;
- выбора режима вычислений с помощью флажков (можно вычислять в любой комбинации такие величины, как сумма, наибольший общий делитель и про изведение двух чисел).

При выборе команды Calc открывается окно сообщений с результатами.

Вариант 7. Создать меню с командами Begin, Help, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Begin открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой input;
- метку типа Label для вывода результата;
- группу из трех переключателей (2, 8, 16) типа RadioButton;
- две кнопки типа Button — Do и OK.

Обеспечить возможность:

- ввода числа в десятичной системе в поле input;
- выбора режима преобразования с помощью переключателей: перевод в двоичную, восьмеричную или шестнадцатеричную систему счисления.

При щелчке на кнопке Do должен появляться результат перевода.

Вариант 8. Создать меню с командами Input size, Choose, Change, Exit. При выборе команды Exit приложение завершает работу. Команда Change недоступна.

При выборе команды Input size открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Size x, Size y;
- кнопку типа Button.

При выборе команды Choose открывается диалоговое окно, содержащее: группу из двух переключателей (Increase, Decrease) типа RadioButton, кнопку типа Button.

Обеспечить возможность ввода значений в поля Size x и Size y. Значения интерпретируются как количество пикселов, на которое надо изменить размеры главного окна (увеличить или уменьшить в зависимости от положения переключателей).

После ввода значений команда Change становится доступной. При выборе этой команды размеры главного окна увеличиваются или уменьшаются на введенное количество пикселов.

Вариант 9. Создать меню с командами Begin, Work, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Begin открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой Input word;
- группу из двух переключателей (Upper case, Lower case) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода слова и выбора режима перевода в верхний или нижний регистр в зависимости от положения переключателей. Вывод результата осуществляется в исходное окно.

Вариант 10. Создать меню с командами Input color, Change, Clear.

При выборе команды Input color открывается диалоговое окно, содержащее:

- группу из двух флажков (Up, Down) типа CheckBox;
- группу из трех переключателей (Red, Green, Blue) типа RadioButton, кнопку типа Button.

Обеспечить возможность:

- выбора цвета с помощью переключателей;
- выбора режима, определяющего, какая область закрашивается; все окно, его верхняя или нижняя половина.

При выборе команды Change цвет главного окна изменяется на заданный. При выборе команды Clear восстанавливается первоначальный вид.

Вариант 11. Создать меню с командами Reverse, Exit, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Reverse открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой input;
- метку типа Label для вывода результата;
- группу из двух переключателей типа CheckBox;
- две кнопки типа Button — Do и OK.

Обеспечить возможность:

- ввода строки символов input;
- выбора режима преобразования с помощью переключателей: перевод в верхний регистр и/или изменение порядка следования на обратный в зависимости от значения переключателей.

При щелчке на кнопке Do должен появляться результат перевода.

Вариант 12. Создать меню с командами Begin, Work, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Begin открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox;
- группу из двух переключателей (First letter, All letters) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода предложения и выбора режима его перевода: начинать прописной буквы каждое слово или перевести все буквы в верхний регистр в зависимости от положения переключателей.

При выборе команды Work открывается диалоговое окно с результатом преобразования.

Вариант 13. Создать меню с командами Input color, Change, Clear.

При выборе команды Input color открывается диалоговое окно, содержащее:

- группу из двух флажков (Up, Down) типа CheckBox;
- группу из трех переключателей (Red, Green, Yellow) типа RadioButton,
- кнопку типа Button.

Обеспечить возможность:

- выбора цвета с помощью переключателей;
- выбора режима, определяющего, какая область закрашивается; все окно, его правая или левая половина.

При выборе команды Change цвет главного окна изменяется на заданный. При выборе команды Clear восстанавливается первоначальный вид.

Вариант 14. Создать меню с командами Input, Calc и Exit.

При выборе команды Input открывается диалоговое окно, содержащее:

- два поля типа TextBox для ввода радиуса основания и высоты цилиндра;
- группу из двух флажков (Объем и Площадь поверхности) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода данных;
- выбора режима с помощью флажков: подсчет объема и/или площади цилиндра.

При выборе команды Calc открывается диалоговое окно с результатами. При выборе команды Exit приложение завершается.

Вариант 15. Создать меню с командами Input size, Choose, Change, Exit. При выборе команды Exit приложение завершает работу. Команда Change недоступна.

При выборе команды Input size открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox с метками Size x, Size y;
- кнопку типа Button.

При выборе команды Choose открывается диалоговое окно, содержащее: группу из двух переключателей типа RadioButton, кнопку типа Button.

Обеспечить возможность ввода значений в поля Size x и Size y. Значения интерпретируются как количество пикселов, на которое надо изменить сдвинуть главное окно (влево или/и вниз в зависимости от положения переключателей).

После ввода значений команда Change становится доступной. При выборе этой команды главного окно сдвигается на заданное количество пикселов.

Вариант 16. Создать меню с командами Input color, Change, Clear.

При выборе команды Input color открывается диалоговое окно, содержащее:

- группу из двух флажков (Up, Down) типа CheckBox;
- группу из трех переключателей (Red, Green, Blue) типа RadioButton, кнопку типа Button.

Обеспечить возможность:

- выбора цвета с помощью переключателей;

- выбора режима, определяющего, какая область закрашивается; все окно, его правая или левая половина.

При выборе команды Change цвет главного окна изменяется на заданный.

При выборе команды Clear цвет окна становится белым.

Вариант 17. Создать меню с командами Reverse, Exit, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Reverse открывается диалоговое окно, содержащее:

- поле ввода типа TextBox с меткой input;
- метку типа Label для вывода результата;
- группу из двух переключателей типа CheckBox;
- три кнопки типа Button — ADD и OK.

Обеспечить возможность:

- ввода строки символов input;
- выбора режима преобразования с помощью переключателей: перевод в верхний регистр и/или изменение порядка следования на обратный в зависимости от значения переключателей.

При щелчке на кнопке ADD должен добавиться введенный текст в поле вывода результата.

Вариант 18. Создать меню с командами Begin, Work, About. При выборе команды About открывается окно с информацией о разработчике.

При выборе команды Begin открывается диалоговое окно, содержащее:

- два поля ввода типа TextBox;
- группу из двух переключателей (First letter, All letters) типа RadioButton;
- кнопку типа Button.

Обеспечить возможность ввода предложения и выбора режима его перевода: начинать прописной буквы каждое слово или перевести все буквы в верхний регистр в зависимости от положения переключателей.

При выборе команды Work открывается диалоговое окно с результатом преобразования.

Вариант 19. Создать меню с командами Input color, Change, Clear.

При выборе команды Input color открывается диалоговое окно, содержащее:

- группу из двух флажков (Up, Down) типа CheckBox;
- группу из трех переключателей (Red, Green, Yellow) типа RadioButton,
- кнопку типа Button.

Обеспечить возможность:

- выбора цвета с помощью переключателей;
- выбора режима, определяющего, какая область закрашивается; все окно, его правая или левая половина.

При выборе команды Change цвет главного окна изменяется на заданный. При выборе команды Clear цвет окна становится белым.

Вариант 20. Создать меню с командами Input, Calc и Exit.

При выборе команды Input открывается диалоговое окно, содержащее:

- два поля типа TextBox для ввода радиуса основания и высоты конуса;
- группу из двух флажков (Объем и Площадь поверхности) типа CheckBox;
- кнопку типа Button.

Обеспечить возможность:

- ввода данных;
- выбора режима с помощью флажков: подсчет объема и/или площади цилиндра.

При выборе команды Calc открывается диалоговое окно с результатами. При выборе команды Exit приложение завершается.

Лабораторная работа 6

Обработка исключительных ситуаций в c#

Язык C# содержит операторы, позволяющие обнаруживать и обрабатывать ошибки, так называемые исключительные ситуации, возникающие в процессе выполнения программы.

Исключительная ситуация (исключение) – это возникновение аварийного состояние, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение.

Исключения в C# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например нажатие определенной клавиши.

Исключение позволяют логически разделить вычислительный процесс на две части: обнаружение аварийной ситуации и ее обработка. Функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место ее возникновения. Это особенно актуально при использовании библиотечных функций н программ, состоящих из многих модулей.

Другим достоинством исключений является то, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение и параметры, поэтому заголовки функций не разрастаются.

Исключения обнаруживаются и обрабатываются в операторе try. Часто используемыми стандартными исключениями являются:

ArithmeticException Ошибка в арифметических операциях или пре-образованиях
ArrayTypeis-matchException Попытка сохранения в массиве элемента несов-местимого типа
DivideByZeroException Попытка деления на ноль
FormatException Попытка передать в метод аргумент неверного формата
IndexOut0fRangeException Индекс массива выходит за границы диапазо-на
InvalidCastException Ошибка преобразования типа
Out0fMemoryException Недостаточно памяти для создания нового объекта
OverFlowException Переполнение при выполнении арифметических операций
StackOverFlowException Переполнение стека

Оператор try-catch

Обработка исключений – это описание реакции программы на подобные события (исключения) во время выполнения программы. Реакцией программы может быть корректное завершение работы программы, вывод информации об ошибке и запрос повторения действия (при вводе данных).

Оператор try-catch. Он имеет следующую структуру:

```
try
{
   //блок кода, в котором возможно исключение
}
catch ([тип исключения] [имя])
{
   //блок кода – обработка исключения
}
```

Выполняется код в блоке try, и, если в нем происходит исключение типа, соответствующего типу, указанному в catch, то управление передается блоку catch. При этом, весь оставшийся код от момента выбрасывания исключения до конца блока try не будет выполнен. После выполнения блока catch, оператор try-catch завершает работу.

Указывать имя исключения не обязательно. Исключение представляет собою объект, и к

нему мы имеем доступ через это имя. Если тип выброшенного исключения не будет соответствовать типу, указанному в catch – исключение не обработается, и программа завершит работу аварийно.

Оператор try-catch также может содержать блок finally. Особенность блока finally в том, что код внутри этого блока выполнится в любом случае, в независимости от того, было ли исключение или нет.

Выполнение кода программы в блоке finally происходит в последнюю очередь. Сначала try затем finally или catch-finally (если было исключение).

Обычно, он используется для освобождения ресурсов. Классическим примером использования блока finally является закрытие файла. Finally гарантирует выполнение кода, несмотря ни на что. Даже если в блоках try или catch будет происходить выход из метода с помощью оператора return – finally выполнится.

Операторы try-catch также могут быть вложенными. Внутри блока try либо catch может быть еще один try-catch. Генерация исключения выполняется с помощью оператора throw.

Оператор throw с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного класса System Exception. Этот объект используется для передачи информации об исключении – обработчику.

Задание

В задании предыдущей работы предусмотреть обработку исключений, возникающих из-за ошибочного ввода пользователя.

Лабораторная работа 7
Абстрактные классы

Задание. В соответствии с вариантом организовать классы. В производных классах не забыть объявление конструкторов и функций вывода данных на печать. Продемонстрировать работу всех методов производных классов. Составить UML-диаграмму иерархии классов.

Варианты

1. Описать абстрактный класс *животное*. Класс должен содержать характеристики животных: название, вид, местообитание, функцию вывода всех данных на экран. На его основе реализовать классы *Млекопитающее*, *Рыба*, *Птица*. Отдельными характеристиками классов являются: для млекопитающих – травоядное, хищник или всеядное; для рыб – морская или пресноводная; для птиц – дикая, домашняя, если дикая перелетная, или нет.
2. Создать абстрактный класс *средство передвижения*. На его основе реализовать классы *самолет*, *машина*, *корабль*. Все классы должны хранить параметры средств передвижения: скорость, расход топлива, наименование производителя, год выпуска, метод вывода на экран всех данных, определения срока службы. Индивидуально для самолета указать высоту и максимальную дальность полета, для машины – объем двигателя, для самолета и корабля – количество посадочных мест, для корабля – водоизмещение.
3. Создать абстрактный класс *линия второго порядка* с полями – коэффициенты уравнения второго порядка. На его основе создать классы окружность, парабола (с методом нахождения директрисы), гипербола, эллипс (с методом нахождения эксцентриситета). Предусмотреть виртуальные методы нахождения центра (вершин или фокусов) линий и функции вывода данных на экран.
4. Описать абстрактный класс *фигура на плоскости*. На его базе создать классы круг, треугольник, прямоугольник. Предусмотреть виртуальные методы создания объектов, вычисление

площади фигур, периметра для треугольника и прямоугольника, длины окружности – для круга.

5. Создать абстрактный класс *правильный многоугольник*. На его основе создать классы треугольник, квадрат, восьмиугольник. Предусмотреть виртуальные методы создания объектов, вычисления их периметра, площади, величины угла.

6. Создать абстрактный класс *правильный многогранник* с полями длина ребра и число ребер. На его основе создать классы тетраэдр, куб, октаэдр (восьмигранник). Предусмотреть виртуальные методы создания объектов, вычисления их площади поверхности и объема.

7. Создать абстрактный класс *вектор*. На его основе создать классы вектор на плоскости, в трехмерном пространстве, в пятимерном пространстве. Предусмотреть виртуальные методы создания объектов, вычисления их длины, вывода на экран их координат.

8. Создать абстрактный класс *человек с* полями год рождения, пол, фамилия, имя. На его основе создать классы школьник (с указанием номера школы и класса), студент (специальность, курс), преподаватель (стаж работы, должность). Предусмотреть виртуальный метод вывода данных на экран и вычисления возраста с указанием у молодежи совершеннолетний или нет, у взрослых пенсионер или нет.

9. Создать абстрактный класс *человек с* полями год рождения, пол, фамилия, имя. На его основе создать классы пациент больницы (с указанием номера заболевания и даты госпитализации), врач (специальность, должность). Предусмотреть виртуальный метод вывода данных на экран и вычисления возраста.

10. Описать абстрактный класс *трехмерная фигура*. На его базе создать классы цилиндр, конус, пирамида. Предусмотреть методы создания объектов, вычисление площади поверхности фигур, объема, площади основания.

11. Описать абстрактный класс *фигура*. На его базе создать классы цилиндр, конус, пирамида. Предусмотреть методы создания объектов, вычисление площади поверхности фигур, объема.

12. Описать абстрактный класс *треугольник*. На его основе создать классы правильный треугольник, равнобедренный треугольник, прямоугольный треугольник. В качестве виртуальных методов выделить функции нахождения периметра, площади, величины углов.

13. Описать абстрактный класс *Призма.* На его основе организовать производные классы треугольная призма, цилиндр, призма с основанием формы квадрата. В качестве виртуальных методов определить функции нахождения площади боковой поверхности, площади основания, площади поверхности призмы, объема призмы.

14. Создать абстрактный класс *Сооружение* с полями адрес, этажность, год постройки с виртуальными методами: Вывод информации на экран и необходимость капитального ремонта. На его основе создать классы *жилой дом* (с полями количество квартир, количество подъездов), архитектурное сооружение (с полями наименование, памятник архитектуры (да, нет), промышленная постройка с полем владелец здания. Необходимость капитального ремонта для жилых зданий 50 лет, для памятников архитектуры 30 лет, для иных сооружений 40 лет.

15. Организовать класс *треугольник*, с полями: длины сторон и методами нахождения периметра треугольника и виртуального метода нахождения площади (по формуле Герона), конструктором. Создать производный класс, переопределив функцию нахождения площади по высоте и основанию.

16. Организовать класс *вектор на плоскости*, с полями - координаты и методами нахождения длины вектора и виртуального метода скалярного произведения векторов, конструктором. Создать производный класс, переопределив функцию нахождения скалярного произведения по фор-

муле через длины векторов.

Лабораторная работа 8

Интерфейсы

Перегрузка операций отношения

Если класс реализует интерфейс *IComparable*, его экземпляры можно сравнивать между собой по принципу больше или меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. Перегрузка операций обычно выполняется путем детегирования, то есть обращения к переопределенным методам *CompareTo* и *Equals*.

Как правило, перегруженный оператор отношения возвращает логическое значение *true* и *false*. Это вполне соответствует правилам обычного применения подобных операторов и дает возможность использовать их перегружаемые разновидности в условных выражениях. Если же возвращается результат другого типа, то тем самым сильно ограничивается применимость операторов отношения.

Задание

Выполнить задание лабораторной работы 7, но вместо абстрактного класса использовать интерфейс. Во всех классах реализовать интерфейс IComparable и перегрузить операции отношения. Объекты отсортировать по самостоятельно выбранному критерию.

Лабораторная работа 9

Коллекции

Задание

Воспользоваться описанием класса лабораторной работы № 7. Для хранения класса создать коллекцию. Вид коллекции выбрать самостоятельно. Написать Windows-приложение для работы с коллекцией, позволяющее:

Добавлять элементы в коллекцию,

Считывать данные из файла,

Записать данные в файл,

Сортировать данные по различным критериям,

Поиск элемента по заданному значению поля,

Вывод элементов коллекции по некоторому критерию,

Удаление элемента из коллекции.

Приложение должно содержать меню и диалоговые окна, обработку исключений в случае ошибок пользователя.

Лабораторная работа 10

Введение в графику

Для вывода линий, геометрических фигур, текста и изображений необходимо создать экземпляр класса Graphics, описанного в пространстве имен System Drawing.

Существует три способа создания объекта данного класса.

1. Ссылку на объект Graphics получают из параметра PaintEventArgs, передаваемого в обработчик события Paint, возникающего при необходимости прорисовки формы или элемента управления:

```
Private voig Form1_Paint (object sender, PaintEventArgs e)
{ Graphics g=e.Graphics;      // использование объекта
```

```
}
```

2. Использование метода CreateGraphics, описанного в классах формы и элемента управления:

```
Graphics g;
g= this.CreateGraphics;
```

3. Создание объекта с помощью объекта-потомка Image. Этот способ используется для изменения существующего изображения:

```
Bitmap bm = new Bitmap (“d: \\picture.bmp”);
Graphics g  = Graphics.FromImage ( bm);
```

После создания объекта типа Graphics его можно использовать для вывода линий, геометрических фигур, текста и изображений. Основными объектами, которые при этом применяются, служат:

Pen – рисование линий и контуров геометрических фигур
Brush – заполнение областей,
Font – вывод текста,
Color – выбор цвета.

В примере приводится текст программы для вывода на форме линии, эллипса и текста.

```
Using System;
Using System.Drawing;
Using System.Windows.Forms;
Namespace WindowsApplication1
{
        public partial class Form1 : Form
        {
        public Form1( ) { InitializeComponent( );}
        private void Form1_Paint ( object sender, PaintEventArgs e )
        using ( Graphics g = e.Graphics)
        { using ( Pen pen = new Pen (Color.Red) )
{ g.DrawLine (pen, 0, 0, 200,100);
                    g.DrawEllipse (pen, new Rectangle(50, 50, 100, 150) );
                  }
                   string s = “Sample Text”;
                   Font font = new Font( “Arial”, 18);
                   float x =100.0;
                   float y =20.0;
                  g.DrawString (s, font, brush, x, y);
                  font.Dispose( );
                  brush.Dispose( );
        }
    }
  }
}
```

Графические объекты потребляют системные ресурсы, поэтому для них рекомендуется использовать метод Dispose – освобождение ресурсов.

Варианты

Варианты 1 – 4. Написать Windows-приложение, которое выполняет анимацию изображения.

Создать меня с командами: Показать изображение, Выбор, Анимация, Стоп, Выход.

При выборе команды Показать изображение в центре экрана рисуется объект, состоящий из нескольких графических примитивов.

Команда Выбор открывает диалоговое окно, содержащее:

- поле типа TextBox с меткой Speed для ввода скорости движения объекта;
- группу Direction из двух переключателей (Left-Right) типа RadioButton для выбора направления движения;

кнопку типа Button.

По команде Анимация объект начинает движение в выбранном направлении до края окна и обратно с заданной скоростью (эффект движения достигается циклической перерисовкой изображения с одновременным сдвигом всех координат);

По команде Стоп движение останавливается.

Команда Выход завершает работу приложения.

Варианты 5 – 8. Написать Windows-приложение, которое выполняет построение четырех различных графиков сложных функций на одной форме одновременно.

Создать меня с командами: Построение, Очистка, Информация о графике, Выход.

Команда Выбор открывает диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- выбор толщину линии;
- кнопку типа Button.

Варианты 9 – 12. Написать Windows-приложение, которое выполняет построение одного из четырех различных графиков сложных функций по выбору пользователя.

Создать меня с командами: Выбор функции, выбор диапазона, Построение, Очистка, Информация о графике, Выход.

Команда Выбор открывает диалоговое окно, содержащее:

- список для выбора типа графика, содержащих выбор аналитической записи графика типа ListBox;
- выбор диапазона построения графика;
- кнопку типа Button.

Варианты 13 – 16. Написать Windows-приложение, которое выполняет анимацию изображения.

Создать меня с командами: Показать изображение, Выбор, Анимация, Стоп, Выход.

При выборе команды Показать изображение в центре экрана рисуется объект, состоящий из нескольких графических примитивов.

Команда Выбор открывает диалоговое окно, содержащее:

- список для выбора цвета изображения;
- группу Direction из двух переключателей (Up-Down) типа RadioButton для выбора направления движения;
- кнопку типа Button.

По команде Анимация объект начинает движение в выбранном направлении до края окна и обратно;

По команде Стоп движение останавливается.

Команда Выход завершает работу приложения.

Варианты 17 – 20. Написать Windows-приложение, которое выполняет построение четырех различных графиков сложных функций одновременно в одной системе координат различными цветами.

Создать меня с командами: Выбор, Построение, Очистка, Информация о графике, Выход.

Команда Выбор открывает диалоговое окно, содержащее:

- список для выбора цвета графика типа ListBox;
- список для выбора типа графика, содержащих выбор аналитической записи графика типа

ListBox;

- выбор диапазона построения графика;
- кнопку типа Button.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

Самостоятельная работа по дисциплине включает изучение учебной литературы, подготовка к выполнению лабораторных заданий и отчетов по их выполнению, а также подготовку к тестированию по теоретическим разделам и непосредственно к экзамену.

Задания к лабораторным работам выдаются заранее, как правило, на первом занятие текущего семестра, и для их успешного их выполнения необходимо предварительное освоение теоретического материала и разбор, приведенных на лекции примеров программ, проработка алгоритма решения разобранных задач и составление собственных алгоритмов. Для этого наряду с конспектами можно воспользоваться учебно-методическим обеспечением для самостоятельной работы, указанным в рабочей программе, и самопроверкой с помощью тестовых заданий, размещенных там же.

Для подготовки к выполнению лабораторных работ и повторения, усвоения (изучения пропущенного) теоретического материала студентам рекомендуется самостоятельно организовать по месту проживания дополнительное рабочее место, оборудованное персональным компьютером, подключённым к сети Интернет, и установленным программным обеспечением, необходимым для разработки программ и указанном в рабочей программе. Общие методические рекомендации по составлению программ на языке программирования дополнительные рекомендации по каждой теме представлены в предыдущих разделах.

В отчете по выполнению индивидуального варианта заданий должны содержатся следующие сведения: формулировка задания, входные и выходные данные, текст программы, тестовые (контрольные) значения входных данных и рассчитанные выходные данные.

Итоговый контроль – экзамен проводится на основании перечней вопросов, представленных в рабочей программе.

Экзамен проводится по билетам. Билет включает два теоретических вопроса и задачу. Ответы на поставленные вопросы студент дает после предварительной подготовки. Преподаватель имеет право задать дополнительные вопросы, если ответ дан неполный или затруднительно однозначно оценить ответ.

Подготовка к экзамену заключается в изучении и тщательной проработке студентом конспектов по всем видов занятий в соответствии с перечнем вопросов к экзамену, представленном в рабочей программе дисциплины. При подготовке рекомендуется использовать конспекты лекций, рекомендованную в рабочей программе литературу, ЭВМ и все теоретические знания, и практические навыки, полученные во время проведения всех видов занятий.

ЛИТЕРАТУРА

1 Павловская, Т. А. C#. Программирование на языке высокого уровня [Текст] : учеб. : рек. Мин. обр. РФ / Т. А. Павловская. - СПб. : Питер, 2007. - 432 с. : рис. - (Учебник для вузов). - Библиогр.: с. 425 . - Алф. указ.: с. 427 . - ISBN 978-5-91180-174-8 (в пер.) :

2 Тузовский, А. Ф. Объектно-ориентированное программирование : учебное пособие для прикладного бакалавриата / А. Ф. Тузовский. — М. : Издательство Юрайт, 2018. — 206 с. — (Серия : Университеты России). — ISBN 978-5-534-00849-4. — Режим доступа : www.biblio-online.ru/book/BDEEFB2D-532D-4306-829E-5869F6BDA5F9.

3 Зыков, С. В. Программирование. Объектно-ориентированный подход : учебник и практикум для академического бакалавриата / С. В. Зыков. — М. : Издательство Юрайт, 2018. — 155 с. — (Серия : Бакалавр. Академический курс). — ISBN 978-5-534-00850-0. — Режим доступа : www.biblio-online.ru/book/E006A65E-B936-4856-B49E-1BA48CF1A52F.

4 Казанский, А. А. Программирование на visual c# 2013 : учебное пособие для прикладного бакалавриата / А. А. Казанский. — М. : Издательство Юрайт, 2018. — 191 с. — (Серия : Бакалавр. Прикладной курс). — ISBN 978-5-534-00592-9. — Режим доступа : www.biblio-online.ru/book/95E1CB2C-3044-46D4-A89B-F4FB2E4275DE.

СОДЕРЖАНИЕ