Основы программирования на языке C#

Темы занятий

- Классы и объекты
- Конструкторы класса
- Инкапсуляция, Наследование, Полиморфизм
- Свойства, Индексаторы
- Перегрузка операторов
- Сериализация
- Делегаты, События
- Библиотека классов (DLL)
- Сортировка встроенными средствами
- Многопоточные приложения
- Создание оконных приложений
- Оконные приложения. Создание графики

Занятие 1. Темы

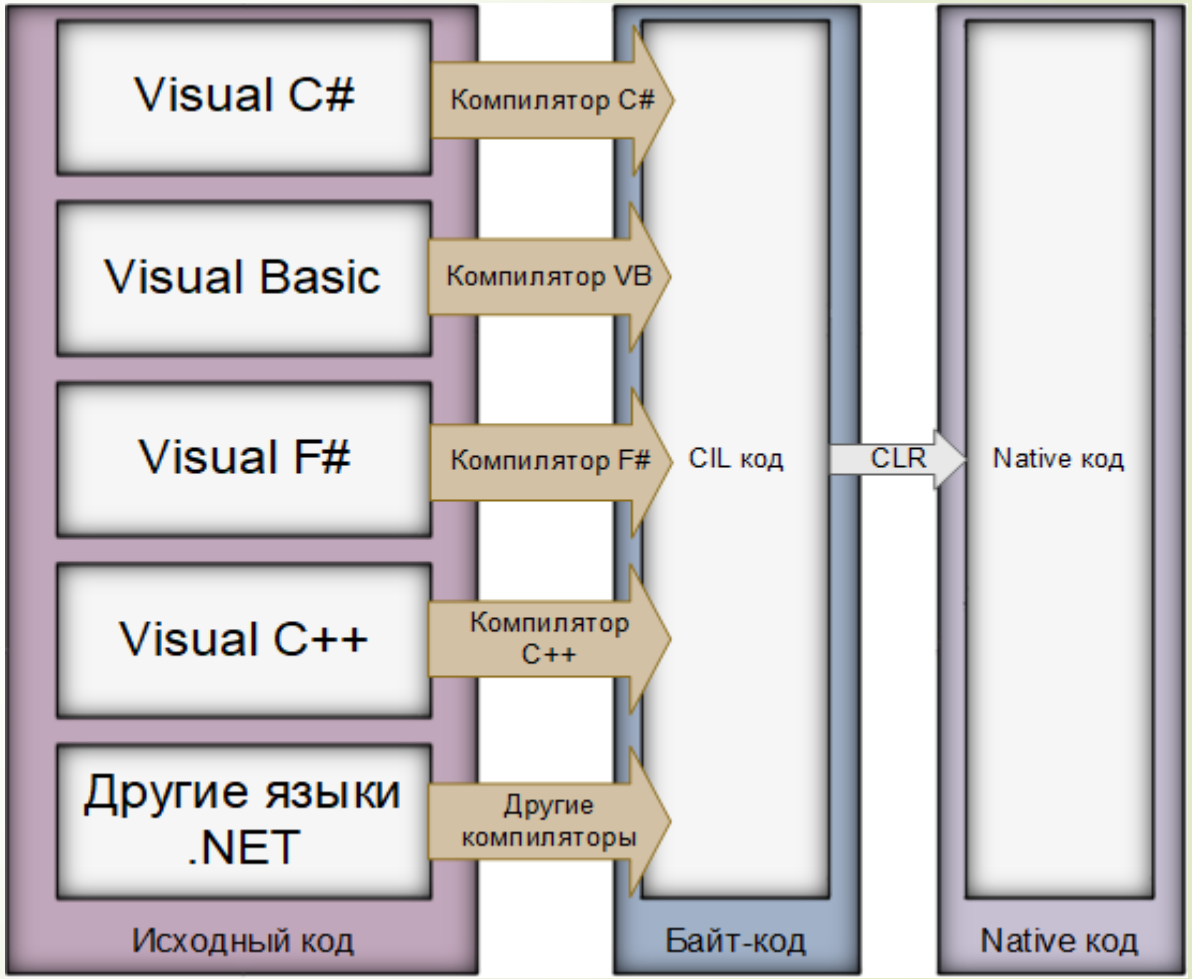
- C# и .NET
- Вывод/вывод данных
- Преобразование данных
- Тип данных bool
- Передача параметров в функцию.
- Передача параметров по ссылке
- Работа с текстовыми файлами
- Генерация случайных чисел
- Массивы
- Обработка исключений

C# и .NET

- C# — объектно-ориентированный язык программирования. Разрабатывается корпорацией Microsoft как язык для создания приложений для платформы Microsoft .NET Framework.
- .NET Framework — программная платформа компании Microsoft. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.
- .NET Framework представляет собой исполняющую среду и обширную библиотеку базовых классов.
- Использование виртуальной машины CLR избавляет разработчиков от необходимости заботиться об особенностях аппаратной части.
- Виртуальная машина CLR также сама заботится о базовой безопасности, управлении памятью и системе исключений, избавляя разработчика от части работы.

C# и .NET

- Программа для .NET Framework, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (CIL). Получившийся код называется сборкой (assembly).
- Получившийся байт-код либо исполняется виртуальной машиной Common Language Runtime (CLR), либо транслируется утилитой NGen.exe в исполняемый код для конкретного целевого процессора.



C++ и C#

- C# является объектно-ориентированным языком. Вся программа строится как взаимодействие различных объектов. Вне объектов ничего не может быть.
- Как и в C++ в C# существуют пространства имен и их подключение с помощью директивы **using**.
- В общеязыковой среде CLR, в которой выполняется программа на C# реализована автоматическая сборка мусора. А это значит, что в большинстве случаев не придется, в отличие от C++, заботиться об освобождении памяти

Вывод/вывод данных

- Для операций ввода/вывода в C# используются методы класса **Console.**
- Класс **Console** находятся в пространстве имен **System** и для удобства работы с ними следует указать на использование этого пространства с помощью команды **using**.
- Вывод данных на экран осуществляется с помощью метода **Write()** или **WriteLine()**
- Ввод данных с клавиатуры осуществляется с помощью метода **Read() или ReadLine()**

Вывод данных

- Вывод данных на консоль осуществляется с помощью метода **Write()**, которому в качестве параметра в двойных кавычках передают текст, который должен быть выведен на экран.
- В методе **Write()** можно соединять текст и переменные с помощью оператора **+**
- Метод **WriteLine()** выводит текст на экран и переводит курсор на новую строку.

```
using System;

namespace CA_TestCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

Ввод данных

- Для ввода данных с клавиатуры используется метод **ReadLine()**
- Метод **ReadLine()** не принимает параметров и возвращает строку введенную с клавиатуры как набор символов (string).
- Это возвращаемое значение можно сохранить в переменную с помощью оператора присваивания и затем использовать в программе.

```
static void Main(string[] args)
        {
            string name;
            Console.WriteLine("What is your name?");
            name = Console.ReadLine();
            Console.WriteLine("Hello, " + name);
        }
```

Преобразование данных

- Для того чтобы преобразовать полученный из файла или клавиатуры текст к другому типу данных можно использовать различные методы класса **Convert**

```
using System;

namespace CA_TestCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int age;
            Console.WriteLine("What is your age?");
            age = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

Тип данных bool

- В языке C# есть логический тип данных – bool.
- Тип bool может принимать только два значения: true (истина) и false (ложь)
- Все операции сравнения двух величин — вещественных и целых переменных или константы с переменной возвращают в качестве результата тип bool.
- Переменную типа bool можно передавать в качестве условия в операторы цикла и ветвления. Если значение переменной типа bool равно false значит условие не выполняется. Иначе выполняется.

Пример:

```
int n;
bool b=true;

while (b)
{
    n = Convert.ToInt32(Console.ReadLine());
     if (n==0)
         b=false;
}
```

Работа с текстом. Тип данных string

- Для работы с текстовой информацией в C# существует тип **string**, являющийся псевдонимом класса **String**, который предоставляет методы для манипуляции строками.
- Переменная типа string представляет собой ссылку на область в памяти, в которой размещены символы.
- Для создания новой строки можно использовать один из конструкторов класса String или напрямую присвоить строке значение, текст в двойных кавычках.
- Объект String является неизменяемым (immutable). При любых операциях над строкой, которые изменяют эту строку, создается новая строка.
- Строки можно соединять оператором конкатенации +.
- Если в конкатенации со строкой соединяется не строка, то она будет преобразована к строке

```
static void Main(string[] args)
{
  string greet, full, name;

  greet = "Hello, ";

  Console.WriteLine("What is your name?");
  name = Console.ReadLine();
  full = greet + name;
  Console.Write(full);
  Console.WriteLine("! Your name has " +
  name.Length + " letters.");
}
```

Класс String

- Основные элементы класса String для работы над своим содержимым:

Элемент	Действие
Length	свойство, которое возвращает длину строки
Split()	разбивает строку на массив строк, с помощью разделителя
Concat()	объединяет строки
Join()	соединяет массив строк в одну строку с разделителем
Compare()	сравнивает две строки
Строка[инд]	строка как массив символов, по индексу можно получить символ
Insert()	вставляет в строку подстроку
IndexOf()	находит индекс первого вхождения подстроки в строку
StartsWith()	определяет, начинается ли строка с подстроки
Replace()	заменяет в строке одну подстроку на другую
Trim()	удаляет начальные и конечные пробелы
Substring()	возвращает подстроку, начиная с определенного индекса
ToLower()	переводит все символы строки в нижний регистр

Передача параметров в функцию

- При передаче в функцию параметров базовых типов (int, float и т.д.) они передаются по значению, т.е. их значения копируются, а изменение этих переменных внутри функции никак не влияет на передаваемые переменные.
- При передаче в функцию объектов и массивов они передаются по ссылке, т.е. в функцию передаются их адреса, и их изменение внутри функции меняет сами переданные переменные.
- Если нужно чтобы параметры простых типов также были переданы по ссылке и их изменение внутри функции изменило сами передаваемые переменные, то их нужно передавать используя ключевые слова **ref** или **out**.

Ключ. слово	Значение
out	Значение параметра должны присваиваться вызываемым методом, он передается по ссылке. Если данному параметру в вызываемом методе значения не присвоено, компилятор сообщит об ошибке
ref	Значение первоначально присваивается вызывающим кодом и при желании может быть изменено в вызываемом методе. Параметр передаётся по ссылке. Если параметру в вызываемом методе значение не присвоено, никакой ошибки нет.
params	Позволяет передавать переменное количество аргументов как единый логический параметр. Метод может иметь только один модификатор params, которым должен быть помечен последний параметр метода.

Передача параметров по ссылке

```
static void changed(ref int a)
{
    a = a + 5;
}

static void Main(string[] args)
{
   int a=3;
   Console.WriteLine("\na before = " + a);
   changed(ref a);
   Console.WriteLine("a after = " + a);
}
```

```
static void changed(out int a, int b)
{
    a = b + 5;
}

static void Main(string[] args)
{
   int a=3;
   Console.WriteLine("\na before = " + a);
   changed(out a, a);
   Console.WriteLine("a after = " + a);
}
```

Работа с текстовыми файлами

- Для работы с текстовыми файлами также используются методы **WriteLine()** и **ReadLine()**, но те которые находятся в классах **StreamReader** (для чтения) и **StreamWriter** (для записи).
- Классы **StreamReader** и **StreamWriter** находятся в пространстве имен **System.IO**
- Нужно создать объект на основе такого класса, при создании указав путь к файлу.
- После использования файл нужно закрыть с помощью метода Close()

```
using System;
using System.IO;

namespace CA_TestCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamWriter sw=new StreamWriter("file.txt");
            sw.WriteLine("Some text in file");
            sw.Close();
        }
```

Генерация случайных чисел

- Генерация случайных чисел осуществляется с помощью класса **Random.**
- Нужно создать объект этого класса и вызвать у него метод **Next()**
- Метод Next() принимает два числа, минимум и максимум, и возвращает случайное целое число находящееся в этом диапазоне.

```
using System;
namespace CA_TestCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;
            Random rnd = new Random();
            i = rnd.Next(10, 99);
            Console.WriteLine("i = " + i);
         }
    }
}
```

Массивы

- Массивы в C# задаются следующим образом: **тип[] название**.
- Размер массива задается с помощью оператора **new**: int[] arr = new int[n];
- Можно инициализировать массив значениями: int[] arr = new int[] { 1, 3, 5, 7, 9 };
- Двумерный массив указывается с помощью запятой, а в операторе new указывается количество строк и столбцов. int[,] matrix = new int[3, 4]; matrix[1, 2] = 3;
- С помощью цикла **foreach** можно пройти по всем элементам массива.

```
int i, n;
Console.WriteLine("Array length?");
n = Convert.ToInt32(Console.ReadLine());
int[] arr = new int[n];

for (i=0; i<n; i++)
    arr[i] = i*10;

foreach (int val in arr)
      Console.Write(val + " ");
Console.WriteLine();
```

Обработка исключений

- Ошибки времени выполнения (**Runtime Exceptions**) это ошибки, которые невозможно отследить в момент компиляции, и которые проявляются уже в процессе работы программы (например в случае если пользователь ввел недопустимые данные)
- Для обработки таких ошибок используется конструкция **try-catch**.
- После ключевого слова **try** в фигурных скобках пишется код, который потенциально может вызвать ошибку.
- После блока **try** идет блок **catch**, у которого в круглых скобках указано ошибки какого рода он ловит, а после этого в фигурных скобках, код который будет выполнен если произошла ошибка.
- После одного блока **try** может быть несколько разных блоков **catch**, которые ловят разные ошибки.
- После блока/блоков **catch** может быть также блок **finally**, содержимое которого выполняется вне зависимости от того была ли ошибка в блоке **try**.

Пример обработки исключений

```
static void Main(string[] args)
        {
           int age=0;
            try
            {
                Console.WriteLine("What is your age?");
                age = Convert.ToInt32(Console.ReadLine());
            }
            catch (FormatException ex)
            {
                age = 0;
            }
            finally
            {
                if (age > 65)
                    Console.WriteLine("Hello pensioner");
                else
                    Console.WriteLine("Hello worker");
            }
    }
```

Занятие 2. Темы

- Объектно-ориентированное программирование
- Классы и объекты
- Сокрытие данных (инкапсуляция)
- Конструкторы класса
- Свойства
- Индексаторы
- Перегрузка операторов
- Переопределение метода ToString
- Статические элементы класса
- Обращение объекта к самому себе

Процедурно-ориентированные языки

- C, Pascal, FORTRAN и другие сходные с ними языки программирования относятся к категории процедурных языков.
- Каждый оператор процедурного языка является указанием компьютеру совершить некоторое действие, например принять данные от пользователя, произвести с ними определенные действия и вывести результат этих действий на экран.
- Программы, написанные на процедурных языках, представляют собой последовательности инструкций, последовательность функций, которые вызывают друг друга.

Недостатки процедурного подхода:

- Процедурный подход не позволяет в достаточной степени упростить сложные программы
- Неконтролируемый доступ к данным. Есть неограниченность доступа функций к глобальным данным.
- Разделение данных и функций плохо отображает картину реального мира.

Объектно-ориентированное программирование

- Идея ООП - объединить данные и действия, производимые над этими данными, в единое целое.
- Три кита объектно-ориентированного подхода:
  1. Инкапсуляция
  2. Наследование
  3. Полиморфизм
- В ООП вводится новое понятие – **класс**. Класс содержит в себе:
  1. поля (данные)
  2. методы (функции для работы с данными)
- **Класс** является по сути пользовательским типом данных. После объявления класса, можно создавать переменные типа этого класса. Такая созданная переменная называется **объектом** (экземпляром) класса.
- Используя имя объекта, можно обращаться к его **полям** и **методам**.
- Типичная программа на языке C# состоит из совокупности объектов, взаимодействующих между собой посредством вызова методов друг друга.

Класс и объект

Объявление класса:

```
class firstClass
{
  private int info;
  public void setInfo(int n)
  {
    info = n;
  }
 public void printInfo()
 {
    Console.WriteLine("info = " + info);
 }
}
```

- Доступ к полям и методам класса осуществляется через конкретный объект этого класса.
- Для того чтобы обратиться к полю или методу, нужно использовать операцию точка (.), связывающую метод или поле с именем объекта.
- Переменная объявленная на основе класса является ссылкой, то есть содержит адрес памяти где хранятся данные объекта.
- Для создания объекта из класса необходимо использовать оператор **new**. Оператор **new** выделяет память для хранения объекта и возвращает адрес.

Создание объекта из класса и вызов его методов:

```
static void Main(string[] args)
{
   firstClass fc;
   fc = new firstClass();
   fc.setInfo(23);
   fc.printInfo();
   firstClass fc2 = new firstClass();
}
```

Сокрытие данных (инкапсуляция)

- Ключевой особенностью объектно-ориентированного программирования является возможность сокрытия данных.
- Данные и методы заключены внутри класса и защищены от несанкционированного доступа извне.
- Если необходимо защитить какие-либо данные, то их помечают ключевым словом **private**. Такие данные доступны только внутри класса.
- Данные, описанные с ключевым словом **public**, напротив, доступны за пределами класса.

```
class secondClass
{  private int privInf;

   public int pubInf;

   public void printInfo()
  {
    Console.WriteLine("info = " + privInf);
  }
};
```

```
static void Main(string[] args)
{
    secondClass sc = new secondClass();
    sc.privInf = 5;       //ошибка! нет доступа
    sc.pubInf =12;
}
```

Конструкторы класса

- Конструктор — это метод класса, выполняющийся автоматически в момент создания объекта.
- С помощью конструктора удобно инициализировать поля объекта автоматически в момент его создания.
- У конструктора есть несколько особенностей, отличающих его от других методов класса.
- Имя конструктора в точности совпадает с именем класса. Таким образом, компилятор отличает конструкторы от других методов класса.
- Конструктор класса должен иметь модификатор доступа public.
- У конструктора не существует возвращаемого значения. Конструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой конструктор мог бы возвратить значение.
- Конструкторы бывают двух видов:
  1. Конструктор по умолчанию (не принимает параметров)
  2. Параметризованный конструктор (со списком параметров)

Использование конструктора

```
class firstClass {
  private int info;


   public firstClass(){
       info = 0;
   }
   public void setInfo(int n) {
       info = n;
   }
   public void printInfo() {
 Console.WriteLine("info = " + info);
   }
}
```

```
static void Main(string[] args)
{
    firstClass fc = new firstClass();
    fc.printInfo();
}
```

Деструктор класса

- Деструктор — это метод класса, выполняющийся автоматически в момент уничтожения объекта.
- В деструкторе обычно производится освобождение памяти, которая была выделена в процессе работы объекта.
- Имя деструктора совпадает с именем конструктора, но в начале имени деструктора добавляется также специальный символ ~ (тильда), чтобы их различать.
- У деструктора также не существует возвращаемого значения. Деструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой он мог бы возвратить значение.

Объектно-ориентированное программирование в C#

- В C# реализованы все принципы объектно-ориентированного программирования: Наследование, инкапсуляция и полиморфизм.
- В большинстве случаев, работа с объектами идет напрямую, а не через указатель, поэтому доступ к полям и методам класса осуществляется с помощью оператора точки (.)
- Модификатор доступа (public, protected, private) указывается отдельно для каждого поля и метода класса.
- Если модификатор доступа не указан, то по умолчанию будет private.

Пример

- Полином от одной переменной это математическая функция заданная следующей формой:

$$a_n * x^n + * a_{n-1} * x^{n-1} + a_{n-2} * x^{n-2} \dots a_2 * x^2 + a_1 * x^1 + a_0 * x^0 = 0$$

где $a_n \dots a_0$ - коэффициенты полинома, x - переменная. Пример : $7x^4 + 10x^2 + 5 = 0$

- Напишем класс для работы с полиномами. Поскольку полином однозначно определяется своими коэффициентами, то в нашем классе достаточно хранить только их.
- Для хранения набора данных одного типа разумно использовать массив.
- Длину этого массива тоже следует хранить внутри класса, она будет говорить о степени полинома.
- Пусть конструктор без параметров создает пустой массив.
- В конструктор с параметрами передадим массив коэффициентов для создания и заполнения массива внутри объекта.

Класс для хранения полинома


```
class Polynom
{
   int[] coeffs;
   int size;

   public Polynom()
   {
      size = 0;
      coeffs = null;
   }

   public Polynom(int[] co)
   {
      size = co.Length;
      coeffs = new int[size];

      for (int i = 0; i < size; i++)
          coeffs[i] = co[i];
    }
}
```

```
static void Main(string[] args)
{
   int[] mas1 = new int[] { 7, 0, 10, 0, 5 };
   int[] mas2 = new int[] { 1, 6, 4, 5, 8 };

   Polynom poly1 = new Polynom(mas1);
   Polynom poly2 = new Polynom(mas2);
   Polynom poly3 = new Polynom();
}
```

Свойства

- Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают простой доступ к полям класса.
- Объявление свойства имеет следующий синтаксис:
  [модификатор_доступа] возвращаемый_тип название
- Определение свойства содержит блоки **get** и **set**. В блоке **get** происходит возврат значения поля, а в блоке **set** его установка. Параметр **value** представляет передаваемое значение.
- Среда Visual Studio может сама сгенерировать для поля свойство. Для этого на поле нужно вызвать контекстное меню и выбрать:
  Оптимизация кода -> Инкапсулировать поле.

Пример свойств класса

```
class Car
   {
      private int number;
      public int Number
      {
         get { return number; }
         set { number = value; }
      }
     private string vendor;
     public Car()
     {
        number = 0;
        vendor = "vaz";
     }
     public void show()
     {
        Console.WriteLine("num: "+number+"\nven:"+vendor);
     }
   }
```

```
static void Main(string[] args)
{
      Car mycar=new Car();
      mycar.Number = 123;
       mycar.show();
}
```

Индексаторы

- Индексатор это разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках.
- Объявление индексатора имеет следующий синтаксис:
  модификатор_доступа возвращаемый_тип this [список_параметров]

```
class Car
   {
      private int[] mas;
      public int this[int i]
      {
        get { return mas[i]; }
        set { mas[i] = value; }
      }
      …
}
```

```
static void Main(string[] args)
{
      Car mycar=new Car();
      mycar[2] = 123;
       mycar.show();
}
```

Перегрузка операторов

- Для того чтобы использовать обычные операции с определенными пользователями типами используется перегрузка операторов.
- Перегрузка операторов позволяет определить или переопределить действия которые будут выполняться конкретным оператором применительно к объекту заданного класса.
- Перегрузить можно только те операторы, которые уже определены в C# (например сложение, деление, проверка на равенство).
- Перегрузка оператора заключается в написании специального статического метода внутри класса, который будет вызван, при обращении к оператору.
- Данный метод состоит из следующих частей
  1. Ключевые слова **public static**
  2. Возвращаемый тип
  3. Ключевое слово **operator**
  4. Символ операции (например +)
  5. Список аргументов, заключенный в скобки

Пример: *public static Polynom operator +(Polynom op1, Polynom op2)*

Переопределение метода ToString()

- Для того чтобы установить вид вывода на экран объекта нужно переопределить его метод **ToString**().

```
class Car
   {
       private int number;
       private string vendor;

      public Car(){
          number = 0;
          vendor = "vaz";
       }


public override string ToString(){
            string str;
            str = "num: "+number+"\nven:"+vendor;
            return str;
        }
   }
```

```
static void Main(string[] args)
{
      Car mycar=new Car();
      Console.WriteLine(mycar)
}
```

Статические элементы класса

- Элементы класса: поля, методы и конструкторы могут быть определены с использованием ключевого слова **static**. В этом случае будет существовать только один такой элемент для всех объектов данного класса.
- Статические поля класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение.
- Обращение к статическому элементу происходит через имя класса, а не объекта.
- Класс целиком тоже может быть объявлен как статический. В этом случае он может включать в себя только элементы помеченные ключевым словом static .

```
class Car{
   static int total = 0;
   public Car(){
      total++;
    }

public static int getTotal(){
    return total;
   }
}
```

```
static void Main(string[] args)
{
  Car c1 = new Car();
  Car c2 = new Car();
  Car c3 = new Car();
  Console.WriteLine("Created " + Car.getTotal() + " cars");
}
```

Обращение объекта к самому себе

- С помощью ключевого слова **this** объект может обратиться сам к себе.
- Одно из возможных применений ключевого слова this это разрешение неоднозначности контекста, когда входящий параметр назван так же, как поле данных класса.
- Также с помощью this объект может передать себя в качестве параметра в функцию.

```
class Car {
   private int number;
   private string vendor;

   public Car(int number, string vendor) {
      this.number = number;
      this.vendor = vendor;
   }

   public void show() {
       Console.WriteLine("num: "+number+"\nven:"+vendor);
   }
}
```

Занятие 3. Темы

- Вложенные классы
- Пространства имен
- Сериализация
- Делегаты
- Передача делегата в функцию
- Шаблон «Наблюдатель»
- События
- Шаблон «Наблюдатель» с событиями

Вложенные классы

- В C# классы могут быть вложенными один в другой, т.е. одни классы, можно определять внутри других классов.
- Внутренний класс может получить доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private.
- Чтобы вложенный класс получил доступ к данным внешнего класса, объект внешнего класса нужно передать в качестве аргумента в конструктор вложенного класса.
- По умолчанию внутренний класс имеет модификатор доступа private.
- Если внутренний класс объявлен с модификаторам public, то объект внутреннего класса можно создавать без создания внешнего:
  Outer.Inner obIn = new Outer.Inner();
- Вложенные классы могут быть статическими. Статические вложенные классы позволяют скрыть некоторую комплексную информацию внутри внешнего класса

Вложенные классы

```
class Car {
  private int number; private string vendor;
  private Motor moto;

  public Car(int number, string vendor, int volume, int power){
     this.number = number; this.vendor = vendor;
     moto = new Motor(volume, power);
  }

  public override string ToString() {
     string str;
     str = "num: " + number + "\nven:" + vendor + "\n";
     str += moto.getInfo();
     return str;
   }

  public class Motor {
    private int volume; private int power;

    public Motor(int volume, int power) {
       this.volume = volume; this.power = power;
     }

  public string getInfo() {
     return "volume: " + volume + "\npower: " + power;
    }
   }
 }
```

```
static void Main(string[] args)
{
   Car mycar = new Car(123, "Ford Focus", 60, 24);
   Console.WriteLine(mycar);

   Console.WriteLine("Мотор без авто:");
   Car.Motor m = new Car.Motor(50, 20);
   Console.WriteLine(m.getInfo());
}
```

Пространства имен

- Классы в C# можно объявлять внутри пространств имен (namespace). Пространства имен позволяют логически сгруппировать классы.
- Чтобы указать, что класс находится внутри определенного пространства имен, надо до объявления класса использовать директиву **namespace**, после которой указывается имя пространства: *namespace название { //то что внутри пространства }*
- Помещение классов внутрь пространств имен позволяет избежать конфликта имен между классами. В разных пространствах классы могут иметь одинаковые названия. Принадлежность к пространству позволяет гарантировать однозначность имени.
- Для обращения к классу находящемуся внутри пространства имен отличного от текущего нужно указать его полное имя в виде Пространство.Класс
- Чтобы не указывать каждый раз название пространства его можно подключить с помощью директивы **using**: *using пространство;*
- Платформа .NET Framework использует пространства имен для упорядочения множества имеющихся в ней классов
- Большинство программ на языке C# начинаются с набора директив using, где перечисляются пространства имен, которые будут часто применяться в программе, чтобы в дальнейшем не указывать их при использовании содержащихся в них классов.

Сериализация

- **Сериализация** это сохранение данных объекта в файл в виде линейной последовательности байт.
- Обратная операция – чтение данных из файла в объект, называется **десериализация**.
- В C# объект можно сохранять в файл в следующих форматах:
  - Binary (двоичный, используя пространство System.Runtime.Serialization.Formatters.Binary)
  - Soap (Soap формат из пространства System.Runtime.Serialization.Formatters.Soap (+reference)
  - XML-файл (пространство имен System.Xml.Serialization)
- Для того чтобы объект класса можно было сериализовать нужно пометить класс атрибутом [Serializable] (написать перед объявлением класса).
- Те поля, которые сохранять не требуется, помечаются атрибутом [NonSerialized]
- Для сериализации объекта в файл нужно:
  - Создать файл с помощью класса FileStream из пространства System.IO
  - Создать вспомогательный объект из класса BinaryFormatter или SoapFormatter или XmlSerializer
  - Используя метод Serialize(файл, объект) произвести сериализацию.

Пример сериализации

Сериализуемый объект

```
[Serializable]
    class Person
    {
        protected int id;
        protected string name;

        public Person(int i, string n)
        {
            id = i;
            name = n;
        }

        public void show()
        {
            Console.WriteLine("Person:\n" +
name);
        }
    }
```

Сериализация

```
static void Main(string[] args)
{
   Person p1 = new Person(1, "Vasya");
   FileStream f = new FileStream("SerFile.bin", FileMode.Create);
   BinaryFormatter bf = new BinaryFormatter();
   bf.Serialize(f, p1);
   f.Close();
}
```

Десериализация

```
static void Main(string[] args){
  Person p1;
  FileStream f = new FileStream("SerFile.bin", FileMode.Open);
  BinaryFormatter bf = new BinaryFormatter();
  p1 = (Person)bf.Deserialize(f);
  f.Close();
  p1.show();
}
```

Делегаты

- Делегат это объект, указывающий на метод или список методов.
- После того как делегат создан и снабжен необходимой информацией, он может динамически вызывать методы, на которые указывает.
- Объявление делегата: **[атрибуты][спецификаторы] delegate тип имя ([параметры])**
- Для того чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

```
class firstClass {
  private int info;

   public void func1(string str) {
      Console.WriteLine("I'm F1 = " + str);
   }
 public void func2(string str) {
      Console.WriteLine("I'm F2 = " + str);
   }
}
```

```
//объявляем делегат
public delegate void myDel(string str);

static void Main(string[] args)
{
    firstClass fc = new firstClass();
    myDel md = new myDel(fc.func1);
    md = md + new myDel(fc.func2);
    md("hello!!");
}
```

Передача делегата в функцию

- Пример использования делегата в качестве параметра функции

```
class firstClass {

//Объявление делегата
public delegate int binOp(int a, int b);

//Функция
public int sum(int a, int b){
    return a + b;
}

//Функция  принимающая делегат
public void sumOut(binOp funcDel, int a, int b){
    int s;
    s = funcDel(a, b);
    Console.WriteLine("sum from func = " + s);
 }
}
```

Основная программа

```
static void Main(string[] args)
{
    int a=3, b=9;
    firstClass fc = new firstClass();
    fc.sumOut(fc.sum, a, c);
}
```

Шаблон «Наблюдатель»

- С помощью делегатов можно реализовать функциональность, когда один объект посылает уведомления другим, подписавшимся на него объектам.
- В классе, объект которого высылает уведомления, должны быть:
  - Делегат, к которому будут прикрепляться методы наблюдателей
  - Методы подписки и отписки наблюдателей
  - Метод вызывающий делегат и тем самым отправляющий уведомления
- В классе, объекты которого получают уведомления, должен быть метод подходящий к делегату, который будет вызван при получении уведомления.
- В основной программе будет создан объект отправляющий уведомления, и объекты это уведомление получающие.
- В основной программе будет осуществляться вызов методов объекта, который высылает уведомления:
  - Подписка наблюдателя на уведомление
  - Отписка наблюдателя от уведомления
  - Отправка уведомления

Пример шаблона «Наблюдатель»

```
//объявление делегата
public delegate void myDel(string str);

//Класс уведомляющий
class Informator {
   private myDel md;

   public void add(myDel obs) {
      md = md + obs;
   }

  public void remove(myDel obs) {
     md = md - obs;
 }

public void inform(string info) {
  if (md!=null)
          md(info);
}
}
```

```
//Класс уведомляемый
class Observer
{
 private int num;

 public Observer(int n)
 {
    num = n;
 }

 public void getInfo(string str)
 {
 Console.WriteLine("I'm
observer №"+num +" got info:
"+str);
  }
}
```

```
//Основная программа
static void Main(string[] args)
{
Informator inf = new Informator();

Observer ob1 = new Observer(1);
Observer ob2 = new Observer(2);
Observer ob3 = new Observer(3);

inf.add(new myDel(ob1.getInfo));
inf.add(new myDel(ob2.getInfo));
inf.add(new myDel(ob3.getInfo));

inf.inform(“text");

inf.remove(new myDel(ob2.getInfo));

inf.inform(“new info");
```

События

- Событие — это специальный публичный элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния.
- События построены на основе делегатов: делегат связан с событием и указывает какие методы будут вызваны при возникновении события
- При использовании событий не требуется описывать методы, регистрирующие и удаляющие обработчики. События поддерживают операции += и -=, добавляющие обработчик в список и удаляющие его из списка.
- Событие и связанные с ним методы-обработчики вызывается как обычная функция: имя и в скобках параметры
- Событие объявляется с использованием ключевого слова **event**
- Объявление события: **public event имя_делегата имя_события;**

Шаблон «Наблюдатель» с событиями

```
//объявление делегата
public delegate void myDel(string str);

//Класс уведомляющий
class Informator {

public event myDel myEv;
public event myDel myEv2;

public void inform(string info) {
  if (myEv!=null)
      myEv(info);
}

public void inform2(string info) {
  if (myEv2!=null)
      myEv2(info);
}
}
```

```
//Класс уведомляемый
class Observer
{
 private int num;

 public Observer(int n)
 {
    num = n;
 }

 public void getInfo(string str)
        {
            Console.WriteLine("I'm
observer №"+num +" got info:
"+str);
        }
    }
```

```
//Основная программа
static void Main(string[] args)
{
Informator inf = new Informator();

Observer ob1 = new Observer(1);
Observer ob2 = new Observer(2);
Observer ob3 = new Observer(3);

inf.myEv += new myDel(ob1.getInfo);
inf.myEv += new myDel(ob2.getInfo);
inf.myEv += new myDel(ob3.getInfo);

inf.inform("text");

inf.myEv -= new myDel(ob2.getInfo);

inf.inform("new text");
}
```

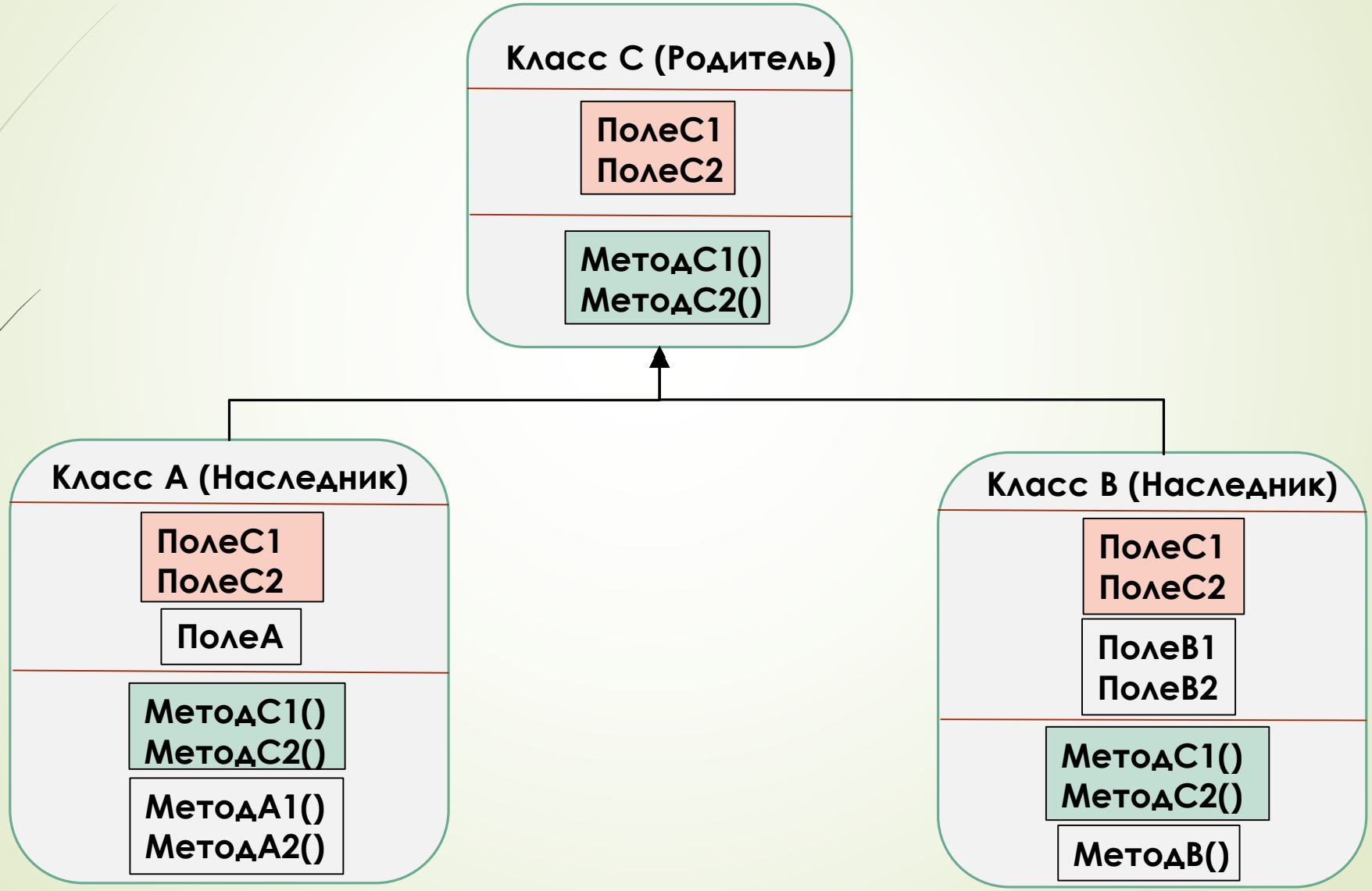
Занятие 4. Темы

- Наследование
- Полиморфизм
- Виртуальные функции
- Абстрактный класс
- IComparable. Сортировка встроенными средствами
- IComparer. Сортировка по нескольким полям

Наследование

- Если есть несколько классов у которых часть полей и методов одинаковая то для того чтобы не дублировать код можно использовать наследование.
- Наследование это создание новых классов, называемых ***наследниками*** или ***производными классами***, из уже существующих или базовых классов. Производный класс получает все возможности базового класса, но может также быть усовершенствован за счет добавления собственных полей и методов.
- Наследование позволяет использовать существующий код несколько раз. Имея написанный и отлаженный базовый класс, необязательно модифицировать его при необходимости внесения в код изменений. Вместо этого можно использовать механизм наследования.
- Наследование позволяет использовать классы, созданные кем-то другим, без модификации кода, просто создавая производные классы, подходящие для частной ситуации.
- Отношение наследования объявляется через двоеточие, сначала приводится класс наследник, потом родительский класс: **class Наследник : Родитель**

Наследование



Пример наследования

Базовый класс

```
class Person
{
   protected int id;
   protected string name;

   public Person()
   {
      Console.WriteLine("I'm person");
      id = 0;
      name = "Anonym";
   }

  public void show()
  {
  Console.WriteLine("Person:“+name);
  }
}
```

Класс наследник 1

```
class Student : Person
{
   private int group;

   public Student()
   {
     Console.WriteLine("I'm student");
     group = 1;
   }

 public new void show()
  {
Console.WriteLine("Student:");
Console.WriteLine("Name: "+name);
Console.WriteLine("Group: " + group);
  }
}
```

Класс наследник 2

```
class Teacher : Person
{
   private string rank;

   public Teacher()
   {
      Console.WriteLine("I'm teacher");
       rank = "Prof.";
   }

  public new void show()
  {
  Console.WriteLine("Teacher:");
  Console.WriteLine("Name:"+name);
  Console.WriteLine("Rank:" + rank);
   }
}
```

Обращение к методам базового класса

- Поля и методы объявленные в родительском классе с модификатором доступа protected или public доступны в классе наследнике, как будто это его собственные поля и методы.
- Если в классе наследника объявлен метод с таким же именем, как в родительском классе, то этот метод скрывает метод родительского класса, и должен быть помечен ключевым словом new перед возвращаемым типом.
- С помощью ключевого слова **base**, можно обращаться к методам родительского класса из класса наследника.
- С помощью ключевого слова **base** можно вызывать родительский конструктор с параметрами из конструктора с параметрами класса наследника.
- Если в классе наследнике определен конструктор без параметров, то в родительском классе тоже должен существовать конструктор без параметров.
- Родительский конструктор без параметров вызывается из конструктора без параметров класса наследника автоматически.

Пример обращения к методам базового класса

Базовый класс

```
class Person
{
   protected int id;
   protected string name;

   public Person(int id, string name)
   {
      Console.WriteLine("I'm person");
      this.id = id;
      this.name = name;
   }

  public void show()
  {
     Console.WriteLine(“ID: “ + id);
     Console.WriteLine(“Name: “ + name);
   }
}
```

Класс наследник

```
class Student : Person {
 private int group;

 public Student(int id, string name, int group) : base(id, name) {
     Console.WriteLine("I'm student");
     this.group = group;
  }

 public new void show() {
     base.show();
     Console.WriteLine("Group: " + group);
  }
}
```

Основная программа

```
static void Main(string[] args)
{
    Student st = new Student(3, "Vasya", 1234);
    st.show();
}
```

Полиморфизм

- Полиморфизм это свойство, которое позволяет одно и то же имя использовать для решения нескольких схожих, но технически разных задач.
- Концепцией полиморфизма является идея "один интерфейс, множество методов".
- В объект базового класса фактически можно записать любой объект наследник.
- Можно создать полиморфную функцию, с одинаковым именем в разных классах наследниках. Выбор конкретной полиморфной функции будет зависеть от того какой именно объект наследник записан в объект базового класса.
- Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.
- Чтобы использовать полиморфизм в C#, необходимо выполнить три условия:
  1. Все классы должны являться наследниками одного и того же базового класса.
  2. Полиморфная функция должна быть объявлена виртуальной (virtual) в базовом классе
  3. Полиморфная функция должна быть объявлена переопределенной (override) в наследниках

Виртуальные функции

- В объектно-ориентированном программировании виртуальная функция это функция класса, которая может быть переопределена в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
- Виртуальные функции это один из важнейших приёмов реализации полиморфизма. Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника.
- Базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.
- Техника вызова виртуальных методов называется "динамическим связыванием".
- Имя метода, использованное в программе, связывается с адресом входа конкретного метода динамически (во время исполнения программы), а не статически (во время компиляции).

Пример полиморфизма

Базовый класс

```
class class Person
   {
      protected int id;
      protected string name;

      public Person()
      {
         Console.WriteLine("I'm person");
         id = 0;
         name = "Anonym";
      }

      public virtual void show()
      {
         Console.WriteLine("Person:\n“
              + name);
      }
   }
```

Класс наследник

```
class Student : Person
   {
      private int group;

      public Student()
      {
         Console.WriteLine("I'm
student");
         group = 1;
      }

      public override void show()
      {
Console.WriteLine("Student:");
         Console.WriteLine("Name: "
+ name);
         Console.WriteLine("Group: "
+ group);
      }
   }
```

```
static void Main(string[] args)
{
    Person[] pArr = new Person[3];

     pArr[0] = new Teacher();
     pArr[1] = new Student();
     pArr[2] = new Student();

     for (i = 0; i < 3; i++)
         pArr[i].show();
}
```

Абстрактный класс

- Абстрактным классом называется базовый класс из которого запрещено создавать объекты.
- Абстрактный класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы.
- Чтобы сделать класс абстрактным необходимо перед его объявлением указать ключевое слово **abstact**.
- Если в классе есть хотя бы одна абстрактная функция то класс должен быть объявлен абстрактным. Абстрактная функция содержит только объявление, без реализации.
- После указания класса как абстрактного создание из него объектов становится невозможным.

Абстрактный базовый класс

```
abstract class Person
{
   public abstract void show();
}
```

Класс наследник

```
class Student : Person
{
    public override void show()
    {
        Console.WriteLine(group);
    }
}
```

Создание объектов

```
static void Main(string[] args)
{
  Student st = new Student();
  Person p = new Person(); //ошибка
}
```

Интерфейсы

- Интерфейсы похожи на абстрактные классы. Они определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.
- Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I: *interface IMovable {...}*
- В интерфейсе можно определить следующие элементы: 1)Методы, 2)Свойства, 3)Индексаторы, 4)События, 5)Статические поля и константы. Но определять переменные внутри интерфейса нельзя.
- При применении интерфейса, как и при наследовании после имени класса указывается двоеточие и затем идет название интерфейса: **class Класс : Интерфейс**.
- Класс не может наследовать от нескольких родительских классов, но может использовать множество интерфейсов.
- Если надо применить в классе несколько интерфейсов, то они перечисляются через запятую

  class Класс : Интерфейс1, Интерфейс2
- Если класс одновременно наследует другой класс и реализует интерфейс, то сначала указывается название базового класса, а потом интерфейса: **class Наследник : Родитель, Интерфейс**
- Если элементы интерфейса не имеют модификаторов доступа, то по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом.
- Интерфейсы могут иметь не только определения методов и свойств, но и их реализацию по умолчанию, которая используется, если класс, использующий данный интерфейс, их не реализует.
- Если класс применяет интерфейс, то он должен реализовать все методы интерфейса не имеющие реализации по умолчанию. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный и содержать абстрактное объявление этих методов.
- С помощью интерфейсов, так же как с помощью наследования можно реализовать полиморфизм.

Шаблон «Наблюдатель» с интерфейсом

```
//интерфейс
interface IObserver
{
   void getInfo(int i);
}
```

```
//класс уведомляемый
class Observer1 : IObserver
{
   int num;

   public Observer1(int num) {
      this.num = num;
   }

   public void getInfo(int i) {
          Console.WriteLine("Я 1 тип
          №"+num+" получил "+i);
   }
}
```

```
//класс уведомляющий
class Informator
{
   int curNum;
   IObserver[] obs = new
   IObserver[10];

  public Informator() {
     curNum = 0;
  }

public void addObserver(IObserver o)
 {
    obs[curNum] = o;
    curNum++;
 }

 public void sendInfo(int info) {
     for (int i = 0; i < curNum; i++)
          obs[i].getInfo(info);
  }
}
```

```
class Observer2 : IObserver {
   int num;

   public Observer2(int num) {
      this.num = num;
   }

   public void getInfo(int i) {
          Console.WriteLine("Я 2 тип
          №"+num+" получил "+i);
   }
}
```

```
static void Main(string[] args)
{
   Observer1 ob1 = new Observer1(1);
   Observer2 ob2 = new Observer2(2);
   Observer1 ob3 = new Observer1(3);

    Informator inf = new Informator();
    inf.addObserver(ob1);
    inf.addObserver(ob2);
    inf.addObserver(ob3);

    inf.sendInfo(55);
}
```

Сортировка встроенными средствами

- В C# массивы можно сортировать с помощью функции Sort находящейся в классе Array. Но если для встроенных типов сортировка происходит автоматически, то чтобы такая сортировка стала возможной для массивов пользовательских типов (объектов) необходимы дополнительные действия:
  - Пользовательский класс должен реализовать интерфейс **IComparable**.
  - В пользовательском класса нужно переопределить метод **CompareTo**.
- Метод **CompareTo** сравнивает текущий объект с объектом, который передается в качестве параметра. На выходе он возвращает целое число, которое может иметь одно из трех значений:
  - Меньше нуля - текущий объект меньше объекта, который передается в качестве параметра.
  - Равен нулю - оба объекта равны.
  - Больше нуля - текущий объект больше объекта, передаваемого в качестве параметра.

Пример сортировки и IComparable

```
class Car : IComparable {
   private int number;
   private string vendor;

   public Car(int number, string vendor) {
      this.number = number;
      this.vendor = vendor;
    }

   public int CompareTo(object o) {
       Car c = (Car)o;
        if (number < c.number) return -1;
            else return 1;
    }

   public void show() {
      Console.WriteLine("num:
"+number+"\nven:"+vendor);
    }
 }
```

```
static void Main(string[] args)
{
   Car[] cars = new Car[3];

   cars[0] = new Car(1, "BMW");
   cars[1] = new Car(3, “Lada");
   cars[2] = new Car(2, “Toyota");

   for (i=0; i< cars.Length; i++)
       cars[i].show();

    Array.Sort(cars);

    for (i = 0; i < cars.Length; i++)
        cars[i].show();
}
```

Сортировка по разным полям

- Если массив пользовательских объектов нужно сортировать по нескольким критериям, тогда можно воспользоваться интерфейсом **IComparer**:
  - Для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий интерфейс IComparer.
  - В этом вспомогательном классе нужно переопределить метод **Compare**. сравнивающий два объекта.
  - Объект этого вспомогательного класса нужно передать в стандартный метод сортировки массива в качестве второго параметра.
- Интерфейс IComparer находится в пространстве имен System.Collections.

Пример сортировки и IComparer

```
class Car {
   private int number;
   private string vendor;
   public Car(int number, string vendor) {
      this.number = number;
      this.vendor = vendor;
    }
 public class compareByNumber : IComparer {
    public int Compare(object ob1, object ob2){
        Car c1 = (Car)ob1;
        Car c2 = (Car)ob2;
        if (c1.number < c2.number) return -1;
            else return 1;
         }
     }
 public class compareByVendor : IComparer {
    public int Compare(object ob1, object ob2){
         Car c1 = (Car)ob1;
         Car c2 = (Car)ob2;
         return c1.vendor.CompareTo(c2.vendor);
     }
   }
}
```

```
static void Main(string[] args)
{
   Car[] cars = new Car[3];

   cars[0] = new Car(1, "BMW");
   cars[1] = new Car(3, “Toyota");
   cars[2] = new Car(2, “Lada");

   for (i=0; i< cars.Length; i++)
       cars[i].show();

 Array.Sort(cars, new Car.compareByNumber());
    for (i = 0; i < cars.Length; i++)
        cars[i].show();

Array.Sort(cars, new Car.compareByVendor());
    for (i = 0; i < cars.Length; i++)
        cars[i].show();
}
```

Занятие 5. Темы

- Библиотека классов (DLL)
- Обобщения (Generics)
- Коллекции
- Класс ArrayList. Динамический массив

Библиотека классов (DLL)

- Библиотека классов это отдельный файл с расширением dll, который содержит классы, которые можно подключить и многократно использовать в других проектах.
- В .Net Framework есть много готовых библиотек классов, но можно также создавать и свои библиотеки.
- Для создания библиотеки следует при разработке проекта в среде Visual Studio.NET выбрать тип проекта Class Library (библиотека классов).
- Библиотека классов не содержит метода Main() поскольку она не запускается напрямую, а только подключается для использования в другом проекте.
- После компиляции библиотеки классов в папке Debug или Release будет файл dll.
- Открыв dll файл с помощью программы ILDasm.exe можно получить полную информацию о библиотеке.
- Для использования библиотеки в проекте необходимо подключить на неё ссылку с помощью команды Project • Add Reference (Добавить ссылку) и выбрать файл dll.
- Также в проекте необходимо подключить то пространство имен, с которым была создана библиотека.

Обобщения (Generics)

- Обобщения или generics позволяют уйти от жесткого определения используемых в классе типов. Если есть несколько похожих классов, которые различаются только типом какого-то поля, то можно написать вместо них один обобщенный класс.
- Для объявления обобщенного класса после его имени в треугольных скобках указывается универсальный параметр, который затем будет замещен на конкретный тип: ***class ИмяКласса<T>***
- При создании объекта из обобщенного класса в треугольных скобках указывается конкретный тип, который будет подставлен вместо универсального параметра: ***Класс<Тип> объект = new Класс<Тип>(параметры)***
- С помощью ключевого слова **where** можно ограничить универсальный параметр: конкретным классом и его наследниками, любым классом (class), или структурой (struct): **class ИмяКласса<T> where T : ограничитель**
- Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. В этом случае универсальный параметр добавляется в объявление метода перед списком параметров: ***public возвращаемыйТип метод<T>(T параметр)***
- При вызове обобщенного метода после его имени в угловых скобках указывается, какой тип будет подставлен на место универсального параметра: ***объект.метод<Тип>(параметры)***

Пример использования обобщений

```
class Car
{
  int number;
  string vendor;
  int maxS;

public Car(int number,
string vendor, int maxS)
{
 this.number = number;
 this.vendor = vendor;
 this.maxS = maxS;
 }

public override string
ToString()
{
  return "Car N." +
  number + " " +
  vendor + ",
  max speed: " + maxS;
 }
}
```

```
class Person
{
  int id;
  string name;

 public Person(int id,
string name)
 {
   this.id = id;
   this.name = name;
 }

 public override string
ToString()
 {
   return "Person:\n" +
"ID: " + id + "\nName:
" + name;
  }
}
```

```
class MyList<T>
{
   T[] arr;
   int total;

public MyList(int size)
{
   arr = new T[size];
   total = 0;
}

public void add(T item) {
  if (total < arr.Length)
 {
   arr[total] = item;
   total++;
  }
}

public void showAll()
 {
  for (int i=0; i<total; i++)
   Console.WriteLine(arr[i]);
 }
}
```

```
static void Main(string[] args)
{
  MyList<Car> carList = new MyList<Car>(5);
 carList.add(new Car(123, "BMW", 100));
 carList.add(new Car(432, "Lada", 80));
 carList.add(new Car(654, “Ford", 70));
 carList.add(new Car(873, "Toyota", 90));
 carList.showAll();

 MyList<Person> peList = new MyList<Person>(3);
 peList.add(new Person(1, "Вася"));
 peList.add(new Person(2,"Петя"));
 peList.add(new Person(3,"Коля"));
 peList.add(new Person(4, "Оля"));
 peList.showAll();
}
```

Коллекции

- Для хранения наборов данных предназначено такое встроенное средство языка как массивы. Однако их не всегда удобно использовать, потому, что они имеют фиксированную длину. Эту проблему в C# решают коллекции.
- Коллекции являются классами используемыми для гибкого хранения набора данных. Кроме гибкого размера, они также реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.
- Классы коллекций находятся в пространстве имен System.Collections
- Коллекции в C# делятся на следующие типы:
  - Классы System.Collections.Generic (универсальная коллекция, все элементы одного типа)
  - Классы System.Collections.Concurrent (потокобезопасные коллекции для нескольких потоков)
  - Классы System.Collections (хранят элементы как объекты типа Object)

Основные классы коллекций

System.Collections

Класс	Описание
ArrayList	динамический массив объектов
Hashtable	набор элементов, где каждый элемент имеет ключ - хеш-код
Queue	очередь, элементы обрабатываются в порядке поступления
Stack	стек, последний поступивший обрабатывается первым

System.Collections.Generic

Класс	Описание
Dictionary<TKey,TValue>	набор пар «ключ-значение», которые упорядочены по ключу
List<T>	список объектов, доступных по индексу
LinkedList<T>	реализация связного списка
SortedList<TKey,TValue>	набор пар "ключ-значение", упорядоченных по ключу
Queue<T>	очередь, элементы обрабатываются в порядке поступления
Stack<T>	стек, последний поступивший обрабатывается первым

Класс ArrayList. Динамический массив

- Класс ArrayList позволяет создать динамический массив, автоматически увеличивающий свой размер при добавлении новых элементов.
- По умолчанию при добавлении первого элемента в массив ArrayList создается массив из 4 элементов. Если при добавлении нового элемента в массив оказывается, что он уже переполнен, то размер массива удваивается.
- Массив ArrayList предназначен для хранения объектов типа object. Но поскольку все типы в C# являются потомками класса object, то фактически каждая ячейка ArrayList может содержать элемент любого типа.
- Доступ к элементу выполняется по индексу, однако необходимо явным образом привести полученную ссылку к целевому типу. int a = (int)arrl[0]
- Класс ArrayList находится в пространстве имен System.Collections.

Основные методы класса ArrayList

Имя	Описание
Capacity	Емкость массива (количество элементов, которые могут в нём храниться)
Count	Фактическое количество элементов массива
Add	Добавление элемента в конец массива
Clear	Удаление всех элементов из массива
IndexOf	Поиск первого вхождения элемента в массив (возвращает индекс найденного элемента)
Insert	Вставка элемента по индексу в заданную позицию (остальные элементы сдвигаются)
Remove	Удаление первого вхождения заданного элемента из массива
RemoveAt	Удаление элемента из массива по заданному индексу (остальные элементы сдвигаются)
Reverse	Изменение порядка следования элементов на обратный
Sort	Сортировка элементов массива или его части

Пример использования ArrayList

```
static void Main(string[] args)
{
    int i;

    ArrayList al = new ArrayList();
    for (i = 1; i <= 5; i++)
      al.Add(i);

     Person p1 = new Person(12, "Name");
     al.Add(p1);
     Console.WriteLine("capacity = " + al.Capacity);

     al[2] = "New item";

     al.Insert(0, 22);
     al.RemoveAt(3);

     for (i = 0; i < al.Count; i++)
        Console.WriteLine(al[i]);
}
```

Занятие 6. Темы

- Многопоточные приложения
- Класс Thread
- Передача параметра в поток
- Синхронизация потоков. Оператор lock
- Использование асинхронных делегатов
- Сетевое программирование

Многопоточные приложения

- В C# есть возможность писать многопоточные приложения, то есть такие программы, которые состоят из нескольких потоков, из нескольких частей кода выполняющихся практически одновременно.
- Основная цель создания многопоточный приложений это повышение общей производительности и сокращение времени реакции приложения.
- Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем.
- В однопроцессорной системе каждый поток получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает впечатление одновременной работы нескольких потоков.
- Многопоточное приложение можно написать только в том случае, когда задачи решаемые программой можно распараллелить.
- Недостатки многопоточности:
  - Накладные расходы, связанные переключением между потоками.
  - Проблемы синхронизации данных, возможностью доступа к одним и тем же данным со стороны нескольких потоков.

Класс Thread

- Для работы с несколькими потоками в C# используются объекты класса Thread, который находится в пространстве имен System.Threading.
- При создании объекта-потока ему передается делегат, определяющий метод, не возвращающий значения, выполнение которого выделяется в отдельный поток.
- Основные элементы класса Thread:

Start	Начинает выполнение потока, определенного делегатом.
Sleep	Приостанавливает выполнение текущего потока на миллисекунды
Suspend	Приостанавливает выполнение потока.
Resume	Возобновляет работу после приостановки потока.
Interrupt	Прерывает работу текущего потока
Join	Блокирует вызывающий поток до завершения другого потока
GetHashCode	Возвращает хеш-код для потока
Name	Установка текстового имени потока
IsAlive	Возвращает информацию о том, запущен поток или нет
CurrentThread	Возвращает ссылку на выполняющийся поток

Пример использования класса Thread

```
class Do {
  int[] arr;

  public Do(int[] a)
  {
     int i;
     arr = new int[a.Length];
     for (i = 0; i < a.Length; i++)
     arr[i] = a[i];
   }

  public void sumArr()
  {
    int i, sum = 0;
    for (i = 0; i < arr.Length; i++)
        sum += arr[i];
    Console.WriteLine("I'm thread №" +
Thread.CurrentThread.GetHashCode() + " sum = " + sum);
  }

 public void mulArr()
 {
     int i, mul = 1;
     for (i = 0; i < arr.Length; i++)
        mul *= arr[i];
      Console.WriteLine("I'm thread №" +
Thread.CurrentThread.GetHashCode() + " mul = " + mul);
      } }
```

```
static void Main(string[] args)
{
  int[] arr = new int[] { 1, 2, 3, 4, 5 };
  Do d = new Do(arr);
  Console.WriteLine("Let's start threading");
  Thread t1 = new Thread(new ThreadStart(d.sumArr));
  Thread t2 = new Thread(new ThreadStart(d.mulArr));
  t1.Start();
  t2.Start();
}
```

Передача параметра в поток

- В функцию вторичного потока можно передать параметр, для этого объект потока нужно создать с помощью делегата ParameterizedThreadStart.
- Во вторичный поток можно передать только один параметр типа object, который затем нужно преобразовать к нужному типу.

```
class Do {
  public void sumArr(object ob)
  {
    int i, sum = 0;
    int[] arr = (int[])ob;
    for (i = 0; i < arr.Length; i++)
        sum += arr[i];
    Console.WriteLine("I'm thread №" +
Thread.CurrentThread.GetHashCode() + " sum = " + sum);
     }

  public void mulArr(object ob)
  {
    int i, mul=1;
    int[] arr2=(int[])ob;
    for (i = 0; i < arr2.Length; i++)
      mul *= arr2[i];
    Console.WriteLine("I'm thread №" +
Thread.CurrentThread.GetHashCode() + " mul = " + mul); } }
```

```
static void Main(string[] args)
{
  int[] arr1 = new int[] { 1, 2, 3, 4, 5 };
  int[] arr2 = new int[] { 10, 20, 30, 40, 50 };
  Do d = new Do();
  Thread t1 = new Thread(new
ParameterizedThreadStart(d.sumArr));
  Thread t2 = new Thread(new
ParameterizedThreadStart(d.mulArr));
   t1.Start(arr2);
   t2.Start(arr1);
}
```

Синхронизация потоков. Оператор lock

- При построении многопоточных приложений необходимо гарантировать, что разделяемые потоками данные защищена от возможности их одновременного изменения разными потоками.
- Если один поток будет прерван до того, как завершит свою работу с разделяемыми данными, то второй поток после этого прочитает нестабильные данные.
- Для синхронизации доступа к разделяемым ресурсам в C# используется ключевое слова lock. Блок команд после оператора lock может выполняться одновременно только одним потоков. (Первый поток, который входит в этот блок блокирует его для остальных).
- Формат оператора lock: lock (выражение) блок_операторов. В качестве выражения для блокировки используется или this или специально создаваемый объект.

Пример синхронизации

```
class Do {
  private int[,] matr;

   public Do(int[,] m) {
      int i, j;
      matr = new int[m.GetLength(0), m.GetLength(1)];

     for (i = 0; i < m.GetLength(0); i++)
        for (j = 0; j < m.GetLength(1); j++)
           matr[i, j] = m[i, j];
    }

  public void TDArr(object ob) {
      int i, row, r = 0;

      row = (int)ob;
      for (i = 0; i < matr.GetLength(1); i++) {
              r = r + matr[row, i];
              Thread.Sleep(2000);
          }
       lock (this) {
              matr[0, 0] += r;
          }
        }}
```

```
static void Main(string[] args)
{
    int[,] matr = new int[60, 50];

    for (i = 0; i < matr.GetLength(0); i++)
      for (j = 0; j < matr.GetLength(1); j++)
            matr[i, j] = 1;

     Do d = new Do(matr);
     Thread[] tt = new Thread[matr.GetLength(0)];

     for (i = 0; i < matr.GetLength(0); i++)
      {
        tt[i] = new Thread(new ParameterizedThreadStart(d.TDArr));
        tt[i].Start(i);
       }

      for (i = 0; i < matr.GetLength(0); i++)
        tt[i].Join();
}
```

Использование асинхронных делегатов

- Делегат можно вызвать для выполнения либо синхронно, либо асинхронно.
  - При синхронном вызове выполняется вызов связанной с делегатом функции и остальная программа продолжается только после завершения работы этой функции.
  - При асинхронном вызове делегата для функции создается отдельный поток и основная программа продолжает выполняться не дожидаясь окончания работы функции делегата.
- Используя асинхронные делегаты можно возвращать значения из вторичного потока.
- Делегат вызывается на выполнение асинхронно с помощью метода **BeginInvoke**.
- Метод **BeginInvoke** имеет те же параметры, что и метод, который нужно выполнить асинхронно, а также два дополнительных необязательных параметра. Первый параметр является делегатом AsyncCallback, который ссылается на метод, вызываемый при завершении асинхронного вызова. Второй параметр это объект, который передает данные в метод обратного вызова.
- Метод **EndInvoke** извлекает результаты асинхронного вызова. Список параметров метода EndInvoke включает параметры out и ref метода, который требуется вызвать асинхронно, а также значение IAsyncResult, возвращаемое методом BeginInvoke
- В методе обратного вызова для получения возвращаемого значения и выходных параметров также применяется метод EndInvoke.

Пример асинхронного делегата

```
class Do {
  public int sumArr(int[] arr2, out int res) {
     int i, sum = 0;
     for (i = 0; i < arr2.Length; i++)
           sum += arr2[i];
     res = sum;
     return 1;
   }

  public int mulArr(int[] arr2, out int res) {
    int i, mul = 1;
    for (i = 0; i < arr2.Length; i++)
         mul *= arr2[i];
     res = mul;
     return 2;
   }

  public void Res(IAsyncResult ar) {
    int res, ret;
   AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;

    ret = ad.EndInvoke(out res, ar);
    if (ret==1)
        Console.WriteLine("Результат сложения = " + res);
    else
       Console.WriteLine("Результат умножения = " + res);
       } }
```

```
//объявление делегата
public delegate int AsyncDelegate ( int[] arr, out int res);

static void Main(string[] args)
{
  int[] arr1 = new int[] { 1, 2, 3, 4, 5 };
  int[] arr2 = new int[] { 10, 20, 30, 40, 50 };

Do d = new Do();
  AsyncDelegate ad = new AsyncDelegate(d.sumArr);
  AsyncDelegate ae = new AsyncDelegate(d.mulArr);

   //определение функции обротного вызова
  AsyncCallback callback = new AsyncCallback(d.Res);

 //вызов функций делегата
 IAsyncResult ar = ad.BeginInvoke(arr2, out res, callback,
null);
 IAsyncResult aq = ae.BeginInvoke(arr1, out res, callback,
null);
}
```

Сетевое программирование

- Для обмена данными между компьютерами существует несколько различных протоколов. Один из них это протокол TCP/IP, в котором для подключения нужно знать адрес и порт компьютера к которому происходит подключение. Адрес представляет собой 32-битную структуру для протокола IPv4, 128-битную для IPv6. Номер порта — целое число в диапазоне от 0 до 65535 (для протокола TCP).
- Пара состоящая из адреса и порта называется **socket** (разъем). Фактически socket обозначает точку, через которую происходит соединение по сети двух программ.
- В процессе обмена, как правило, используется два сокета — сокет отправителя и сокет получателя. В программе можно создать «слушающий» сокет (серверный сокет), который будет находиться в цикле ожидания и просыпаться при появлении нового соединения.
- Обмен данными с помощью сокетов осуществляется по клиент-серверной схеме: сервер находится в режиме ожидания и ждет подключения, клиент зная адрес и порт сервера (сокет) подключается к нему и начинает обмен информацией. Клиент посылает серверу запросы, а тот отправляет на них ответы.

Сетевое программирование в C#

В C# для обмена данными с использованием механизма сокетов используется класс **Socket**, который находится в пространстве имен System.NET.Sockets

На сервере:

1. Определяем адрес и порт, создаем объект типа **IPEndPoint.**
2. Создаем сокет (объект **Socket**) с соответствующими параметрами.
3. Связываем сокет с локальной точкой (адресом и портом) методом **Bind** (передаем ему объект **IPEndPoint**)
4. Начинаем прослушивание с помощью метода **Listen.**
5. Когда подключение пришло на сокет его можно получить с помощью метода **Accept** и присвоить новому сокету.
6. Используя методы **Receive** и **Send** полученного сокета считываем и отправляем данные как массив byte.

На клиенте:

1. Определяем адрес и порт, создаем объект типа **IPEndPoint**.
2. Создаем сокет (объект **Socket**) с соответствующими параметрами.
3. Вызываем у созданного сокета метод **Connect**, которому передаем информацию про адрес и порт, к которому нужно подключиться (объект **IPEndPoint**).
4. Используя методы **Send** и **Receive** отправляем и считываем данные как массив byte.

Сетевое программирование в C#

```
static void Main(string[] args){                    //Сервер
   string text;
   int bytes;
   byte[] buffer = new byte[256];

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8005);

  Socket listenSocket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
  try {
     listenSocket.Bind(ipPoint);
     listenSocket.Listen(10);
     Socket handler = listenSocket.Accept();

    bytes = handler.Receive(buffer);
    text = System.Text.Encoding.Unicode.GetString(buffer, 0, bytes);
     Console.WriteLine(text);

    text = text + " sent";
    buffer = Encoding.Unicode.GetBytes(text);
    handler.Send(buffer);
    handler.Shutdown(SocketShutdown.Both);
    handler.Close();
       }
  catch (Exception ex) {
             Console.WriteLine(ex.Message);
   }}
```

```
static void Main(string[] args) {                    //Клиент
  string outText, inText;
  int bytes;
  byte[] buffer = new byte[256];
IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8005);
  Socket socket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

 try {
   socket.Connect(ipPoint);
   outText = Console.ReadLine();
   buffer = Encoding.Unicode.GetBytes(outText);
   socket.Send(buffer);
   buffer = new byte[256];
   bytes = socket.Receive(buffer, buffer.Length, 0);
 inText = System.Text.Encoding.Unicode.GetString(buffer, 0, bytes);
   Console.WriteLine("server sent me: " + inText);

   socket.Shutdown(SocketShutdown.Both);
   socket.Close();
 }
 catch (Exception ex)
 {
     Console.WriteLine(ex.Message);
 }}
```

Занятие 7. Темы

- Создание оконных приложений
- Основные элементы оконного приложения
- Создание многооконной программы
- Диалоговые окна
- Передача данных между окнами
- Табличный вывод данных. dataGridView
- Оконные приложения. Создание графики

Создание оконных приложений

- С помощью C# можно разрабатывать также оконные приложение, где взаимодействие с пользователем осуществляется в графическом режиме, приложение выполняет вывод в отведенную ему прямоугольную область экрана, называемую окном, которое состоит из стандартных элементов, кнопок, надписей, полей для ввода данных, и т.д.
- При работе оконного приложения используется принцип событийного управления. После запуска программа ожидает действий пользователя и реагирует на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется событием.
- При разработке оконного приложения в Visual Studio следует выбрать тип проекта Windows Forms.
- При создании проекта Windows Forms создается шаблон с пустым окном на которое можно добавлять необходимые элементы, перетаскивая их с панели элементов.
- Добавленным элементам можно затем прикреплять методы, которые будут вызваны при наступлении определенных событий.
- Формы (окна) в Visual Studio можно просматривать как в визуальном режиме (конструктор), так и в виде исходного кода.

Основные элементы оконного приложения

Название	Англ.	Описание
Метка	Label	Размещение текста на форме
Кнопка	Button	Основное событие - щелчок мышью
Поле ввода	TextBox	Позволяет вводить и редактировать текст, который запоминается в свойстве Text
Меню	MainMenu	Меню приложения, находящееся в верхней части программы
Флажок	CheckBox	Для проверки, установлен ли флажок, анализируют его свойство Checked
Переключатель	RadioButton	Позволяет выбрать один из нескольких предложенных вариантов
Панель	GroupBox	Используется для группировки элементов на форме
Список	ListBox	Перечень элементов, в котором можно выбрать одно или несколько значений
Таблица	dataGridView	Отображение данных в виде таблицы

Создание многооконной программы

- Оконное приложение может состоять из нескольких окон, вызывающих одно другое.
- Для создания второго окна следует добавить в приложение новый элемент типа «Форма Windows Forms»
- Для отображения окна следует создать экземпляр объекта соответствующей формы, а затем вызвать для этого объекта метод Show.
- Для того чтобы закрыть окно нужно вызывать у объекта-окна метод Close.
- Передать из вторичного окна данные, можно создав в нем открытое свойство и обращаясь к нему из основного окна.

Form1.cs Основная форма

```
Form2 f2;
private void
button1_Click(object sender,
EventArgs e) {
    f2 = new Form2();
    f2.Show();
}
```

Form1.cs Основная форма

```
private void
button2_Click(object sender,
EventArgs e)
{
    MessageBox.Show("Hello!"
+ textBox1.Text +f2.someText);
}
```

Form2.cs Вторичная форма

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

Диалоговые окна

- Диалоговое окно является особой разновидностью окна. У диалогового окна неизменяемые размеры и есть кнопка OK, подтверждающая введенную информацию, и Cancel, отменяющая ввод. При нажатии на любую из этих кнопок окно закрывается.
- Кнопкам на диалоговом окне присваиваются значения с помощью свойства DialogResult.
- Открытие диалогового окна блокирует другие окна программы. Перейти к ним можно только после закрытия диалогового окна.
- Диалоговое окно открывается с помощью метода ShowDialog соответствующего объекта-формы.

```
private void button3_Click(object sender, EventArgs e)
{
  Form3 f3 = new Form3();
  if (f3.ShowDialog() == DialogResult.OK)
     MessageBox.Show("Нажат Ок!");
  else
      MessageBox.Show("Нажат Cancel");
 }
```

Передача данных между окнами

- Поскольку вторичное окно создается как объект внутри основного окна, то передавать данные из вторичного окна в основное можно просто обращаясь к открытым элементам вторичного окна из основного.
- Для передачи данных из основного окна во вторичное нужно изменить конструктор вторичного окна так чтобы он принимал параметры и при создании вторичного окна передавать те параметры, которые должно получить вторичное окно.
- В конструктор вторичного окна можно передавать либо отдельные данные, либо все основное окно целиком с помощью this.

Form1.cs основная форма

```
private string info;

private void button1_Click(object sender,
EventArgs e)
{
   Form2 f2 = new Form2(info);
   f2.Show();
}
```

Form2.cs Вторичная форма

```
public partial class Form2 : Form {
        private string info;
        public Form2(string info)
        {
            InitializeComponent();
            this.info = info;
            textBox1.Text = info;
        }
}
```

Табличный вывод данных. dataGridView

- С помощью элемента dataGridView на форме отображается таблица с данными.
- Данные в dataGridView можно вводить несколькими способами:
  - Ввод с клавиатуры во время работы программы
  - Чтение из подключенной базы данных
  - Программное заполнение с помощью команд заполнения ячеек
- Свойство *ColumnCount* устанавливает число столбцов, *RowCount* – строк.
- Свойство *Columns[номер_столбца].Name* устанавливает имя столбца
- Функция Rows.Add() без параметров добавляет новую пустую строку, с массивом строк в качестве параметра добавляет новую строку и заполняет каждую ячейку строки значениями из соответствующих ячеек массива.
- Свойство Rows[номер_строки].Cells[номер_столбца].Value устанавливает значение соответствующей ячейки в таблице.

Пример заполнения dataGridView

```
public Form2()
{
    InitializeComponent();
    dataGridView1.ColumnCount = 5;
    dataGridView1.Columns[0].Name = "Release Date";
    dataGridView1.Columns[1].Name = "Track";
    dataGridView1.Columns[2].Name = "Title";
    dataGridView1.Columns[3].Name = "Artist";
    dataGridView1.Columns[4].Name = "Album";

    string[] row0 = { "11/22/1968", "29", "Revolution 9",  "Beatles",
"The Beatles [White Album]" };
    string[] row1 = { "1", "2", "3", "4", "5" };

    dataGridView1.Rows.Add(row0);
    dataGridView1.Rows.Add(row1);
    dataGridView1.Rows.Add();
    dataGridView1.Rows[2].Cells[1].Value = «Some text";
}
```

Оконные приложения. Создание графики

- На форму Windows Forms можно выводить не только текстовую информацию, но и графическую. Можно вставлять готовые рисунки, а также рисовать прямо на форме линии и другие геометрические фигуры.
- Для рисования на форме нужно создать объект класса Graphics из пространства имен System.Drawing. Объект класса Graphics можно создать:
  - Получив ссылку на объект Graphics из параметра PaintEventArgs, передаваемого в обработчик события Paint.
  - Использовав метод CreateGraphics, из класса формы или элемента управления.
  - Создав объект с помощью объекта-потомка Image
- Также для рисования используются следующие классы:
  - Pen — ручка. рисование линий и контуров геометрических фигур;
  - Brush — кисть. заполнение областей;
  - Font — вывод текста;
  - Color — цвет
  - Point – точка по которой проходит линия

Основные методы класса Graphics

Имя	Описание	Параметры
DrawLine	Рисовать линию	pen, x1, y1, x2, y2
DrawRectangle	Рисовать прямоугольник	pen, x, y, ширина, высота
DrawEllipse	Рисовать эллипс	pen, x, y, ширина, высота
DrawPolygon	Рисовать многоугольник	pen, массив точек Point
FillRectangle	Заполнить прямоугольник	brush, x, y, ширина, высота
FillEllipse	Заполнить эллипс	brush, x, y, ширина, высота
FillPolygon	Заполнить многоугольник	brush, массив точек Point

Пример рисования на форме

```
public Form4()
{
   InitializeComponent();
   Graphics g = this.CreateGraphics();
   this.Show();
   Pen pen = new Pen(Color.Red);
   Pen pen2 = new Pen(Color.Green, 5);
   SolidBrush brush = new SolidBrush(Color.Yellow);
   Point[] pt = new Point[3] { new Point(10, 10), new Point(300, 10),
new Point(150, 150) };

   g.DrawLine(pen, 0, 0, 200, 100);
   g.DrawEllipse(pen, new Rectangle(50, 50, 100, 150));
   g.DrawRectangle(pen2, 10, 10, 100, 50);
   g.FillEllipse(brush, 40, 40, 120, 60);
   g.FillPolygon(brush, pt);
 }
```

Работа с данными

- Для современных программ характерен большой объем данных с которыми они работают. Для хранения данных были придуманы специальные средства хранения называемые базами данных (БД). В БД данные хранятся в виде связанных между собой таблиц.
- Для создания и манипулирования базами данных используются специальные программные средства называемые Системами управления базами данных (СУБД). Существуют много различных СУБД, например MS SQL Server, MS Access, Oracle, MySQL, DB2, и т.д.
- В Visual Studio можно либо подключиться к отдельно созданной БД, либо создать и использовать БД локально. БД используемые в проекте показаны на вкладке Обозреватель серверов (Server Explorer).
- Для создания локальной БД нужно добавить в проект новый элемент Локальная база данных. Будет создан sdf файл представляющий собой локальную базу данных в СУБД MS SQL Server Compact Edition (MS SQL Server CE)
- После создания или подключения к базе данных можно создавать в ней таблицы и заполнять их данными на вкладке Обозреватель серверов (Server Explorer).

Строка подключения к базе данных

- Для подключения к БД нужно определить строку подключения, в которой нужно записать информацию о базе данных и сервере, к которым нужно подключиться.
- Строку подключения можно записать либо в переменную типа string, либо хранить в файле конфигурации проекта (app.config или web.config) и считывать оттуда.
- При хранении информации о подключении в файле конфигурации эта информация будет доступна для всех файлов проекта.
- Для определения строки подключения в файле конфигурации нужно в пределах узла <configuration> добавить узел <connectionStrings>. В этом узле определяются строки подключения с помощью элемента <add>. Каждая строка подключения имеет название, определяемое с помощью атрибута name.
- Для того чтобы работать с конфигурацией приложения, и считать оттуда строку подключения надо добавить в проект библиотеку System.Configuration.dll и использовать пространство имен System.Configuration.
- Для получения строки подключения из файла конфигурации можно воспользоваться следующей командой:
ConfigurationManager.ConnectionStrings["имя_строки_подключения"].ConnectionString

Пример использования строки подключения

```
//app.config
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

<connectionStrings>
            <add name="ConnectionName"
            connectionString="Data Source=..\..\MyDB.sdf" providerName="Microsoft.SqlServerCe.Client.3.5" />
    </connectionStrings>

</configuration>
```

```
//Program.cs
using System.Configuration;

namespace CA_ADO.NET_Test {
    class Program {
        static void Main(string[] args) {
            string connectionString =
            ConfigurationManager.ConnectionStrings["ConnectionName"].ConnectionString;
            Console.WriteLine(connectionString);
        }
}
```

Работа с базой данных

- Чтобы подключиться к базе данных SQL Server CE, нужно использовать объект класса **SqlCeConnection**. В конструктор объекта SqlCeConnection передается строка подключения к базе данных.
- После создания объекта SqlCeConnection можно открыть подключение к указанной БД его методом **Open()**, а после завершения работы с БД закрыть подключение методом **Close()**.
- После подключения к БД можно выполнять в ней различные команды, такие как чтение данных, добавление, удаление или изменение.
- Для выполнения команд над базой *SQL Server CE* используется объект класса **SqlCeCommand**. В качестве параметров конструктор объекта SqlCeCommand принимает sql-запрос и объект SqlCeConnection, связанный с открытой БД в которой этот запрос нужно выполнить.
- После создания объекта SqlCeCommand нужно вызвать один из его методов для выполнения команды:
  - **ExecuteNonQuery**: выполняет запрос и возвращает количество измененных записей. Подходит для вставки, изменения и удаления данных.
  - **ExecuteReader**: выполняет запрос и возвращает строки из таблицы. Подходит для чтения данных
  - **ExecuteScalar**: выполняет запрос и возвращает одно скалярное значение, например, число.

Язык SQL. SQL-запросы

- В реляционных базах данных, данные хранятся в виде связанных между собой таблиц, и для работы с ними используется структурированный язык запросов SQL (Structured Query Language)
- В языке SQL существуют следующие команды для работы с данными:
  - SELECT - выборка данных: SELECT какие_столбцы FROM из_какой_таблицы WHERE условие
  - INSERT - добавление данных: INSERT INTO в_какую_таблицу (столбец, ...) VALUES (значение,...)
  - UPDATE - изменение данных: UPDATE в_какой_таблице SET столбец1=новое_значение1, столбец2 = новое значение2,... WHERE условие
  - DELETE - удаление данных: DELETE FROM в_какой_таблице WHERE условие
- Для чтения данных из БД используется метод ExecuteReader объекта SqlCeCommand, он возвращает объект SqlCeDataReader, в котором содержатся данные полученные в результате выполнения запроса на выборку данных.
- Свойство HasRows объекта SqlCeDataReader возвращает информацию о том, были ли выбраны какие-нибудь строки.
- Метод Read() считывает одну строку данных, а метод GetValue(индекс) возвращает данные из соответствующего столбца.

Пример работы с базой данных

```
string sqlExpression;
SqlCeCommand command;
SqlCeDataReader reader;
SqlCeConnection connection;

string connectionString =
ConfigurationManager.ConnectionStrings["ConnectionString"].
ConnectionString;

connection = new SqlCeConnection(connectionString);
try {
  connection.Open();

  sqlExpression = "SELECT * FROM People WHERE sex='F'";
  command = new SqlCeCommand(sqlExpression, connection);
  reader = command.ExecuteReader();

  while (reader.Read()) {
     object id = reader["id"];
     object name = reader["name"];
     object family = reader["family"];
     object age = reader["age"];
     object sex = reader["sex"];
      Console.WriteLine(id+" "+name+" "+family+" "+age+" "+sex);
   }
  reader.Close();
```

```
sqlExpression = "INSERT INTO People (name, family, age, sex)
VALUES ('Том', 'Форд', 67, 'M')";
command = new SqlCeCommand(sqlExpression, connection);
int number = command.ExecuteNonQuery();
Console.WriteLine("Добавлено "+ number+" объектов");

sqlExpression = "UPDATE People SET Age=33 WHERE id=2";
command.CommandText = sqlExpression;
command.ExecuteNonQuery();

sqlExpression = "SELECT * FROM People";
command.CommandText = sqlExpression;
reader = command.ExecuteReader();

while (reader.Read()){
  object id = reader. GetValue(0);
  object name = reader. GetValue(1);
  object family = reader. GetValue(2);
  object age = reader. GetValue(3);
  object sex = reader. GetValue(0);
 Console.WriteLine(id+" "+name+" "+family+" "+age+" "+sex);
 }
 reader.Close();
```

База данных и dataGridView

- Данные из таблицы в БД можно отобразить на форме с помощью элемента dataGridView:
  - Создать форму. На форму добавить элемент DataGridView.
  - В DataGridView нажать на треугольник справа -> Choose Data Source -> Add Project Data Source -> Database -> Dataset (Будет добавлен DataSet, BindingSource и TableAdapter)
  - В форме правой кнопкой на DataSet -> Edit in DataSet Designer
  - В DataSet Designer правой кнопкой на DataSet -> Configure -> Advanced Options -> Выбрать Insert, Update, Delete
- Для обновления данных из dataGridView в таблицу в БД по кнопке нужно добавить в метод нажатия на кнопку следующий код: *myTableAdapter.Update(myDataSet)*

n.b. При открытии файла с БД из Visual Studio открывается файл в корне проекта. При запуске программы происходит работа с файлом БД, который находится в папке Debug|Release!!!

Чтобы изменения внесенные в базу при работе программы были сохранены нужно изменить у файла с базой свойство CopyToOutputDirectory на Copy if newer или прописать полный путь до файла БД в корне проекта.