МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ЛАБОРАТОРНОЙ РАБОТЕ
«Делегаты и события в языке C#»

по дисциплине «Технологии программирования»

для студентов специальностей

122 – Компьютерные науки и информационные технологии,

124 – Системный анализ, в том числе для иностранных студентов

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

Харьков

НТУ «ХПИ»

2017

Методические указания к лабораторным работам «Делегаты и события в языке C#» по дисциплине «Технологии программирования» : для студентов специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ, в том числе для иностранных студентов / сост.: Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков. – Харьков : НТУ «ХПИ», 2017. – 40 с. – На рус. яз.

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

Кафедра системного анализа и информационно-аналитических технологий

Введение

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

Понятие типа делегата в языке C# выполняет роль типа указателя на функцию в языках C/C++. На основе типа делегата организуется тип события. Без этих типов невозможно представить разработки эффективных и производительных приложений. Понятие события в языке C++ было введено в рассмотрении для обеспечения технологии визуального проектирования приложений под Windows. В языке C# события можно использовать и при разработке обычных консольных приложений, поскольку событие является элементом класса.

Существенным шагом вперёд по отношению к языку C# является то, что делегат может выполняться как в синхронном, так и в асинхронном режиме, что собственно и закладывает основы параллельного программирования на языке C# для современных многоядерных компьютеров.

Предлагаемые методические указания помогут студентам ознакомиться с указанными типами и освоить основы организации параллельных вычислений на языке C# с использованием делегатов и событий.

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

1. Делегаты и события

События используются для уведомления о разных ситуациях, случившихся в разных частях приложения. Если в приложении генерируется событие, то в других частях приложении можно отреагировать на него исполнением методов, которые называются обработчиками события.

События являются основой для организации работы компонентов и программирования под Windows.

Но механизм событий можно использовать для создания своих приложений.

1.1. Делегаты

Основой для механизма событий являются делегаты.

Делегат выполняет те же действия, что и указатель на функцию, но более безопасными и лучше соответствующими принципам объектно-ориентированного программирования способами.

Делегаты – это особые классы, обеспечивающие работу событий, способные хранить указатели (ссылки) на методы. Любой делегат производится от единого базового класса - *System.MulticastDelegate* с заранее определенным набором членов.

При объявлении делегата указывают сигнатуру вызываемого им метода и тип возвращаемого результата (т.е. прототип метода):

```
public delegate double MyTypeDelegate(double x);
```

В этом примере объявляется тип делегата *MyTypeDelegate*, способный хранить ссылки на методы, которые возвращают результат типа *int* и принимают в качестве параметра значение типа *double.*

Создание экземпляра типа делегата (делегата) позволяет неявно вызывать исполнение метода по ссылке, которую он хранит. В примере показано, как использовать делегаты для статических и нестатических методов.

```
namespace ConsoleApplication1
{
  class MyClass
```

```
 {
 public delegate double MyTypeDelegate(double x);

 public double Metod(double x)
  {
   return Math.Cos(x);
  }
 public static double StaticMetod(double x)
  {
   return Math.Sin(x);
  }
 }


 class Program
 {
  static void Main(string[] args)
   {
     double x = Math.PI;

     MyClass example = new MyClass();

     MyClass.MyTypeDelegate obDelegate;
     obDelegate=new MyClass.MyTypeDelegate(example.Metod);

     Console.WriteLine("x={0} Metod({1})={2}", x, x, obDelegate(x));

     obDelegate=new
                  MyClass.MyTypeDelegate(MyClass.StaticMetod);
     Console.WriteLine("x={0} Metod({1})={2}", x, x, obDelegate(x));
   }
 }
}
```

В примере строка: *MyClass.MyTypeDelegate obDelegate;* объявляет экземпляр *obDelegate* типа делегата *MyClass.MyTypeDelegate*.

Метод, ссылку на который хранит делегат, называют методом-обработчиком делегата. В примере такими являются методы класса *MyClass Metod()* и *StaticMetod()*.

Строка:

obDelegate=new MyClass.MyTypeDelegate(example.Metod);

создаёт объект делегат *obDelegate* и назначает ему нестатический метод объекта *example* в качестве метода-обработчика делегата.

Строка:

obDelegate=new MyClass.MyTypeDelegate(MyClass.StaticMetod);

создаёт объект делегат *obDelegate и н*азначает ему статический метод *StaticMetod()* класса *MyClass* в качестве метода-обработчика делегата.

Вызов метода, ссылку на который хранит делегат осуществляется строкой *obDelegate(x)*, т.е. указывается объект-делегат, за которым в скобках указывается список параметров, передаваемых в метод-обработчик делегата.

По своей реализации класс *System.MulticastDelegate* является многоадресным, т.е. может хранить ссылки на несколько методов, эти возможности наследуют и производные от него, определяемые пользователем делегаты. Такие делегаты ещё называют групповыми.

Для добавления метода-обработчика делегата используется операция + (+=), а для удаления –(-=).

При использовании групповых делегатов накладываются дополнительные ограничения на методы, на которые они могут ссылаться:

- эти методы должны возвращать тип *void*;

- в списке параметров этих методов не должно быть параметров со спецификатором out.

При вызове группового делегата все добавленные делегаты вызываются в том порядке, в котором они были добавлены в групповой делегат.

В примере ниже показано, как использовать групповые делегаты.

```
namespace ConsoleApplication1
{
   class MyClass
   {
      public delegate void MyTypeDelegate(double x, ref double res);

      public void PrintCos(double x, ref double res)
      {
         res = Math.Cos(x);
         Console.WriteLine("cos({0})={1}", x, res);
      }

      public void PrintSin(double x, ref double res)
      {
         res = Math.Sin(x);
         Console.WriteLine("sin({0})={1}", x, res);
      }
   }


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

         MyClass example = new MyClass();
         double x = Math.PI;
         double res=double.NaN;

         MyClass.MyTypeDelegate obDelegate;

         Console.WriteLine("До вызова объекта делегата res={0}",
                                 res);

         obDelegate =new MyClass.MyTypeDelegate(example.PrintSin);
         obDelegate+=new MyClass.MyTypeDelegate(example.PrintCos);
         obDelegate+=new MyClass.MyTypeDelegate(example.PrintSin);

         obDelegate(x, ref res);
```

```
            Console.WriteLine("После вызова объекта делегата res={0}",
                                          res);
            obDelegate -=new MyClass.MyTypeDelegate(example.PrintSin);
            obDelegate(x, ref res);

            Console.WriteLine("После вызова объекта делегата res={0}",
res);
       }
   }
}
```

В примере объявляется объект делегат *obDelegate,* в который добавляются делегаты для методов объекта *example PrintSin(),PrintCos() и PrintSin()*. В качестве методов-обработчиков используются по сути два метода *PrintCos() и PrintSin()*. Каждый из них отвечает указанным выше ограничениям на методы для групповых делегатов.

Строка *obDelegate(x, ref res);* выполняет вызов объекта группового делегата, который к моменту вызова хранит три ссылки на методы-обработчики.

Результат запуска программы представлен на рис.1.1. Из рис.1.1 видно, что в результате вызова делегата сначала была выполнена последовательность вызовов методов: *PrintSin(),PrintCos(), PrintSin(),* что соответствует порядку их добавления в объект делегат, а после удаления из группового делегата ссылки на метод *PrintSin()* (заметим, что она была удалена из конца последовательности ссылок) и повторного вызова делегата была вызвана последовательность функций *PrintSin(),PrintCos().*

```
file:///E:/_WorkDir/_UchProcess/_Учебные курсы/_Курс Технология программирования ...
До вызова объекта делегата res=NaN
sin(3,14159265358979)=1,22460635382238E-16
cos(3,14159265358979)=-1
sin(3,14159265358979)=1,22460635382238E-16
После вызова объекта делегата res=1,22460635382238E-16
sin(3,14159265358979)=1,22460635382238E-16
cos(3,14159265358979)=-1
После вызова объекта делегата res=-1
```

Рис. 1.1.

Отметим, что ограничение о невозможности использовать параметры типа out не запрещает использовать параметры типа ref, но такой параметр после отработки вызова делегата будет хранить значение, установленное последним методом в последовательности ссылок, которые хранит групповой делегат.

Необходимо добавить, что над значениями типов делегатов определены операции сравнения == и !=. Значение операции == даёт значение истинно в том случае, если сравниваемые объекты делегаты имеют совпадающие последовательности делегатов, которые их образуют.

1.2. События

События являются членами класса, которые позволяют уведомлять другие компоненты приложения о том, что случилось с этим классом.

Схема механизма событий включает два этапа:

- формирование события и
- его последующую обработку.

Формирование события связано с анализом какой-либо ситуации (проверкой какого-либо логического условия, определяющего факт свершения реального события) в определенной точке исполнения программы и уведомления заинтересованных компонентов о нём.

Обработка события выполняется каждым компонентом отдельно, что является его реакцией на случившееся событие. Для возможности реагировать на событие компонент должен содержать специальный метод (обработчик события), который должен быть связан с этим событием.

На одно и то же событие в программе могут реагировать более одного компонента и каждый по разному. Моменты возникновения события и его обработка происходят ассинхронно.

Для обработки события из точки программы, в которой возникло событие, в метод-обработчик события могут быть переданы необходимые параметры, описывающие случившуюся ситуацию.

Объявление событий основано на использовании делегатов. Для объявления события необходимо явно указать тип-делегат, который будет использовать объявляемое событие.

Механизм событий предполагает использование следующих программных действий:

- объявления типа делегата, на котором будет основано событие;
- объявления события;
- определения обработчиков события;
- генерации событий;
- динамического подключения и отключения обработчиков событий.

Ниже дан пример программы с использованием событий.

```
namespace DelegateEvent
{
   class Functions
   {
      public delegate double FuncDelegate(double x);

      FuncDelegate[] func = {
                        new FuncDelegate(Math.Cos),
                        new FuncDelegate(Math.Sin),
                        new FuncDelegate(Math.Exp),
                        new FuncDelegate(Math.Log),
                        new FuncDelegate(Math.Log10),
                        new FuncDelegate(Invert),
                       };

      public static double Invert(double x)
      {
         return 1 / x;
      }

      public delegate void NextPoint(int sz, double x, double [] 
func_res);
      public delegate void ErrorState(string text);

      public event NextPoint  EventNextPoint;
      public event ErrorState EventErrorState;
```

```
void PrintError(string text)
{
   Console.Write("{0}",text);
}

void PrintString(int sz, double x, params double[] func_res)
{
   string str_res = x.ToString();
   if (str_res.Length >= 8)
      str_res = str_res.Substring(0, 7);

   Console.Write(str_res.PadRight(sz));

   foreach (double res in func_res)
   {
      str_res =res.ToString();

      if (str_res.Length >=8)
         str_res = str_res.Substring(0, 8);
      if (!double.IsInfinity(res) && !double.IsNaN(res))
         Console.Write(str_res.PadRight(sz));
      else
         EventErrorState("неопр".PadRight(sz));
   }
   Console.WriteLine();
}

public void Table(int sz, double from, double to, double step)
{
   double[] r = new double[func.Length];

   Console.WriteLine("_".PadRight(sz * 7, '_'));
   Console.WriteLine();
   Console.WriteLine("{0}{1}{2}{3}{4}{5}{6}","x".PadRight(sz),
                     "Cos(x)".PadRight(sz),
                     "Sin(x)".PadRight(sz),
                     "Exp(x)".PadRight(sz),
                      "Ln(x)".PadRight(sz),
                      "Lg(x)".PadRight(sz),
```

```
                              "1/x".PadRight(sz));
        Console.WriteLine("_".PadRight(sz*7,'_'));

        for (double x = from; x <= to; x += step)
        {
            int i = 0;
            foreach (FuncDelegate f in func)
              r[i++] = f(x);

            EventNextPoint(sz, x, r);
        }
        Console.WriteLine("_".PadRight(sz * 7, '_'));
    }

    public Functions()
    {
        EventNextPoint = new NextPoint(PrintString);
        EventErrorState = new ErrorState(PrintError);
    }
}


class Program
{
    static void Main(string[] args)
    {
        Functions ob = new Functions();

        Console.WriteLine("Input a,b,step:");

        string src=Console.ReadLine();

        string[] a_b_step = src.Split(new Char[] { ' ', ';' });

        double a = double.Parse(a_b_step[0]);
        double b = double.Parse(a_b_step[1]);
        double step = double.Parse(a_b_step[2]);

        ob.Table(10, a, b , step);
```

```
            Console.ReadKey();
        }
    }
}
```

Программа запрашивает интервал табулирования функций и шаг и выводит таблицу функций.

Работа программы основана на событиях.

Комментарий к программе:

1) Для вызова каждой из семи фукций по указателю используется класс *Functions,* в котором объявляется тип делегата с именем *FuncDelegate:*

```
public delegate double FuncDelegate(double x);
```

2) Определяется массив делегатов (указателей на функции)

```
FuncDelegate[] func = { new FuncDelegate(Math.Cos),
                        new FuncDelegate(Math.Sin),
                        new FuncDelegate(Math.Exp),
                        new FuncDelegate(Math.Log),
                        new FuncDelegate(Math.Log10),
                        new FuncDelegate(Invert),
                      };
```

3) Определяются типы делегатов, на основе которых определяются события:

public delegate void NextPoint(int sz, double x, double [] func_res);- делегат *NextPoint* задаёт прототип для функции обработчика события *EventNextPoint* (следующая точка интервала обсчитана).

Функция, указатель на которую он хранит, принимает в качестве параметров:
int sz – размер поля вывода значения в таблице,
double x – значение переменной *x*, текущей точки интервала, для которой посчитаны значения функций;

params double [] func_res – значения функций в точке *x*, как массив.

```
public delegate void ErrorState(string text);
```

- делегат *ErrorState* задает прототип для функции обработчика состояния ошибки вычисления *EventErrorState* (состояние ошибки вычисления). Функция, указатель на которую он хранит принимает в качестве параметра строку текста *string text*.

4) Объявляются события *EventNextPoint, EventErrorState:*

```
public event NextPoint  EventNextPoint;
public event ErrorState EventErrorState;
```

Для объявления события используется ключевое слово *event* , за которым следует тип делегата, определяющий прототип функций – возможных обработчиков этого события, и затем – идентификатор объекта события.

Каждому объявляемому событию соответствует статический класс, определенный как private. Его назначение - привязывать событие к соответствующему делегату.

Этот класс включает два скрытых метода. Один из методов начинается с приставки add_XXXX, а второй – с приставки remove_XXXX (XXXX – это идентификатор объекта события), которые используются соответственно для добавления и удаления обработчиков события.

5) Определяются функции обработчики событий

```
void PrintError(string text)
{
   Console.Write("{0}",text);
}
```

и

```
void PrintString(int sz, double x, params double[] func_res)
{
   string str_res = x.ToString();
   if (str_res.Length >= 8)
      str_res = str_res.Substring(0, 7);
```

```
    Console.Write(str_res.PadRight(sz));

    foreach (double res in func_res)
    {
       str_res =res.ToString();

       if (str_res.Length >=8)
          str_res = str_res.Substring(0, 8);
       if (!double.IsInfinity(res) && !double.IsNaN(res))
          Console.Write(str_res.PadRight(sz));
       else
          EventErrorState("неопр".PadRight(sz));
    }
    Console.WriteLine();
 }
```

Из примера видно, что функция *void PrintError(string text)* просто выводит строку *text* на консоль.

А функция *void PrintString(int sz, double x, params double[] func_res)* для заданной точки *x* вычисленных значений функций формирует текущую, соответствующую этой точке строку таблицы.

Отметим, что если значение какой-либо функции в точке *x* не определено или бесконечность, то формируется событие *EventErrorState("неопр".PadRight(sz));*

6) Создаются объекты событий и выполняется их инициализация. Эти действия выполняются в конструкторе класса *Functions:*

```
 public Functions()
 {
    EventNextPoint = new NextPoint(PrintString);
    EventErrorState = new ErrorState(PrintError);
 }
     }
```

При создании объекта события в качестве параметра передается функция обработчик события.

Так, для события *EventNextPoint* в качестве обработчика события устанавливается функция обработчик *PrintString,* а для события *EventErrorState – PrintError.*

7) Динамическое подключение и отключение обработчиков событий.

Отметим, что в основе события лежит многоадресный делегат, который позволяет организовывать многоадресные делегаты. Именно это свойство и используется при организации событий.

Т.е. при срабатывании события будет вызван каждый из обработчиков, которые хранит делегат события. Такой способ позволяет сразу нескольким «приемникам событий» получать одно-единственное происшедшее событие.

Для возможностей добавления обработчиков событий (или их удаления) служат операторы += (-=).

Рассмотрим пример. Допустим, мы хотим, чтобы каждая строка формируемой нами таблицы подчеркивалась пунктирной линией. Этого можно добиться, добавив еще один обработчик события:

```
void PrintLine(int sz, double x, double[] func_res)
    {
        Console.WriteLine("-".PadRight(sz * 7, '-'));
    }
```

и определить событию *EventNextPoint* этот обработчик в качестве приемника:

```
EventNextPoint += PrintLine; // добавить в конструктор класса Functions
```

После этого добавления для события *EventNextPoint* будет установлено два приемника (обработчика) этого события, которые сработают в порядке их добавления – сначала выведется строка значений функций для текущей точки *x*, а затем отделяющая её линия.

Добавление обработчиков в событие-делегат называют установкой прослушивания события (организацией прослушивания события).

8) генерация событий

Для построения таблицы функций на заданном интервале используется метод *Table* класса *Functions*:

```
public void Table(int sz, double from, double to, double step)
{
   double[] r = new double[func.Length];

   Console.WriteLine("_".PadRight(sz * 7, '_'));
   Console.WriteLine();
   Console.WriteLine("{0}{1}{2}{3}{4}{5}{6}","x".PadRight(sz),
                     "Cos(x)".PadRight(sz),
                     "Sin(x)".PadRight(sz),
                     "Exp(x)".PadRight(sz),
                     "Ln(x)".PadRight(sz),
                     "Lg(x)".PadRight(sz),
                     "1/x".PadRight(sz));
   Console.WriteLine("_".PadRight(sz*7,'_'));

   for (double x = from; x <= to; x += step)
   {
      int i = 0;
      foreach (FuncDelegate f in func)
        r[i++] = f(x);

      EventNextPoint(sz, x, r);
   }
   Console.WriteLine("_".PadRight(sz * 7, '_'));
}
```

Он строит заголовок таблицы и в цикле от начальной до конечной точки интервала с заданным шагом формирует для неё массив значений функций (что соответствует состоянию следущей точки) и генерирует событие

EventNextPoint(sz, x, r);

Для генерации события указывается объект событие, за которым в круглых скобках передаются параметры, используемые его обработчиком.

То есть, метод *Table* только организует вычисления, а построение самой таблицы на консоли выполняют обработчики событий.

9) Таким образом, реализованый класс *Functions* содержит для построения таблицы метод

public void Table(int sz, double from, double to, double step),

котороый основан на событиях.

Метод *Main* использует его для построения таблицы:

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

   Functions ob = new Functions();

   Console.WriteLine("Input a,b,step:");

   string src=Console.ReadLine();

   string[] a_b_step = src.Split(new Char[] { ' ', ';' });

   double a = double.Parse(a_b_step[0]);
   double b = double.Parse(a_b_step[1]);
   double step = double.Parse(a_b_step[2]);

   ob.Table(10, a, b , step);

   Console.ReadKey();
}
```

Пример работы программы:

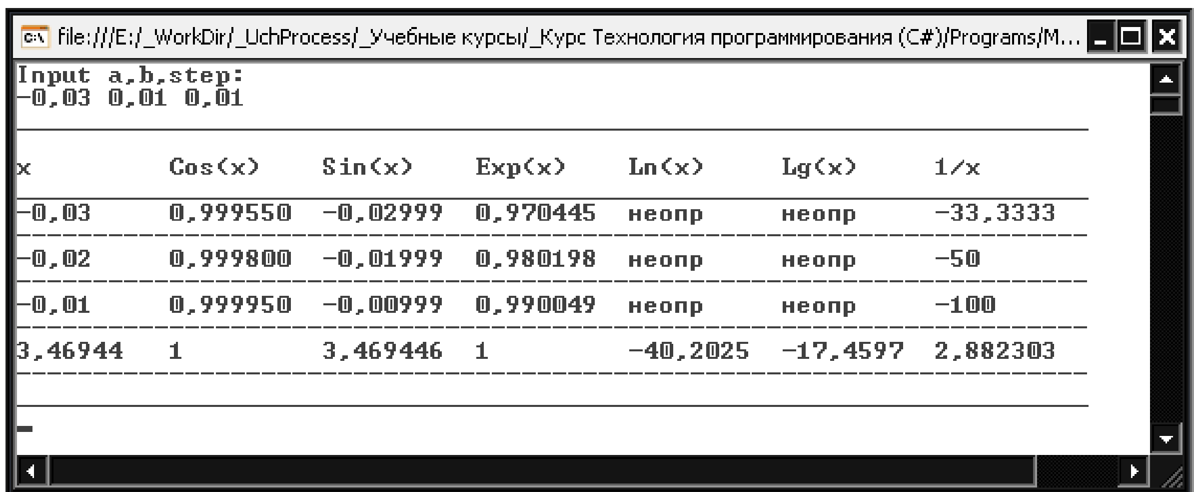

Рис. 1.2.

Из теста видно, что таблица строится и механизм событий работает.

Замечание:

Таблица строится неверно. Обратите внимание на последнюю строку.
Поле *x* должно содержать значение 0, но содержит значение 3,46944,
Точно так же, как и некоторые другие поля значений функций неправильные (например exp()).

Ошибку дает оператор *string str_res = x.ToString();*

Вопрос почему?

Ответ: потому что

```
string str_res = x.ToString();
```

формирует представление в строке в формате с плавающей точкой

```
if (str_res.Length >= 8)
   str_res = str_res.Substring(0, 7);
```

а затем мы берем начало строки – первые 7 символов, т.е. неправильно получаем строковое представление.

Выход из ситуации новая – версия программы

```
namespace DelegateEvent
{
   class Functions
   {
      public delegate double FuncDelegate(double x);

      FuncDelegate[] func = { new FuncDelegate(Math.Cos),
                              new FuncDelegate(Math.Sin),
                              new FuncDelegate(Math.Exp),
                              new FuncDelegate(Math.Log),
                              new FuncDelegate(Math.Log10),
                              new FuncDelegate(Invert),
                            };

      public static double Invert(double x)
      {
         return 1 / x;
      }

      public delegate void NextPoint( int sz,
                                      double x,
                                      params double [] func_res);
      public delegate void ErrorState(string text);

      public event NextPoint  EventNextPoint;
      public event ErrorState EventErrorState;

      void PrintError(string text)
      {
         Console.Write("{0}",text);
      }

      void PrintString(int sz, double x, double[] func_res)
      {
         string str_res =Math.Round(x,3).ToString();
         if (str_res.Length >= 8)
            str_res = str_res.Substring(0, 7);

         Console.Write(str_res.PadRight(sz));
```

```
  foreach (double res in func_res)
  {
     str_res =Math.Round(res,3).ToString();
     if (str_res.Length >=8)
        str_res = str_res.Substring(0, 8);
     if (!double.IsInfinity(res) && !double.IsNaN(res))
        Console.Write(str_res.PadRight(sz));
     else
        EventErrorState("неопр".PadRight(sz));
  }
  Console.WriteLine();
}

void PrintStringNatur(int sz, double x, double[] func_res)
{
  Console.Write("{0:E2} ",x);

  foreach (double res in func_res)
    Console.Write("{0:E2} ",res);

  Console.WriteLine();
}

void PrintLine(int sz, double x, double[] func_res)
{
  Console.WriteLine("-".PadRight(sz * 7, '-'));
}

public void Table(int sz, double from, double to, double step)
{
  double[] r = new double[func.Length];

  Console.WriteLine("_".PadRight(sz * 7, '_'));
  Console.WriteLine();
  Console.WriteLine("{0}{1}{2}{3}{4}{5}{6}","x".PadRight(sz),
                    "Cos(x)".PadRight(sz),
                    "Sin(x)".PadRight(sz),
                    "Exp(x)".PadRight(sz),
```

```
                              "Ln(x)".PadRight(sz),
                              "Lg(x)".PadRight(sz),
                              "1/x".PadRight(sz));
        Console.WriteLine("_".PadRight(sz*7,'_'));

        for (double x = from; x <= to; x += step)
        {
          int i = 0;
          foreach (FuncDelegate f in func)
            r[i++] = f(x);

          EventNextPoint(sz, x, r);
        }
        Console.WriteLine("_".PadRight(sz * 7, '_'));
    }

    public Functions()
    {
        EventNextPoint = new NextPoint(PrintString);

        EventNextPoint += PrintStringNatur;

        EventNextPoint += PrintLine;

        EventErrorState = new ErrorState(PrintError);
    }
}


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

        Functions ob = new Functions();

        Console.WriteLine("Input a,b,step:");

        string src=Console.ReadLine();
```

```
            string[] a_b_step = src.Split(new Char[] { ' ', ';' });

            double a = double.Parse(a_b_step[0]);
            double b = double.Parse(a_b_step[1]);
            double step = double.Parse(a_b_step[2]);


            ob.Table(11, a, b , step);

            Console.ReadKey();
        }
    }
}
```

и её тест приведенный на рис. 1.3.

```
file:///E:/_WorkDir/_UchProcess/_Учебные курсы/_Курс Технология программирования (C#)/Programs/Metod2/Del...
Input a,b,step:
-0,05 0,02 0,01
______________________________________________________________________________

x           Cos(x)     Sin(x)     Exp(x)     Ln(x)      Lg(x)      1/x
______________________________________________________________________________
-0,05       0,999      -0,05      0,951      неопр      неопр      -20
-5,00E-002 9,99E-001 -5,00E-002 9,51E-001 NaN NaN -2,00E+001
------------------------------------------------------------------------------
-0,04       0,999      -0,04      0,961      неопр      неопр      -25
-4,00E-002 9,99E-001 -4,00E-002 9,61E-001 NaN NaN -2,50E+001
------------------------------------------------------------------------------
-0,03       1          -0,03      0,97       неопр      неопр      -33,333
-3,00E-002 1,00E+000 -3,00E-002 9,70E-001 NaN NaN -3,33E+001
------------------------------------------------------------------------------
-0,02       1          -0,02      0,98       неопр      неопр      -50
-2,00E-002 1,00E+000 -2,00E-002 9,80E-001 NaN NaN -5,00E+001
------------------------------------------------------------------------------
-0,01       1          -0,01      0,99       неопр      неопр      -100
-1,00E-002 1,00E+000 -1,00E-002 9,90E-001 NaN NaN -1,00E+002
------------------------------------------------------------------------------
0           1          0          1          -40,203    -17,46     2,882303
3,47E-018 1,00E+000 3,47E-018 1,00E+000 -4,02E+001 -1,75E+001 2,88E+017
------------------------------------------------------------------------------
0,01        1          0,01       1,01       -4,605     -2         100
1,00E-002 1,00E+000 1,00E-002 1,01E+000 -4,61E+000 -2,00E+000 1,00E+002
------------------------------------------------------------------------------

______________________________________________________________________________
```

Рис. 1.3.

Для проверки правильности вывода таблицы введен дополнительный обработчик события EventNextPoint, который выводит строку таблицы без преобразования.

2. Режимы выполнения делегатов

2.1. Синхронный режим

Делегат может быть запущен на выполнение как в синхронном, так и асинхронном режимах.

```
public delegate int BinaryOp(int x, int y);

public static int Add(int x, int y)
    {
        Console.WriteLine("Add() выполняется в потоке {0}  ",
                Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(5000);
        return x + y;
    }

private void button_Click(object sender, EventArgs e)
    {// иллюстрация синхронного вызова делегата
        Console.WriteLine("Main() выполняется в потоке {0}  ",
                Thread.CurrentThread.ManagedThreadId);

        // создаем объект делегата с указателем на Add()
        BinaryOp bop = new BinaryOp(Add);

        Console.WriteLine("Выполняем делегат в синхронном
режиме:");
        Console.WriteLine("10+5={0}", bop(10, 5));
        Console.WriteLine("Main() завершился");
    }
```

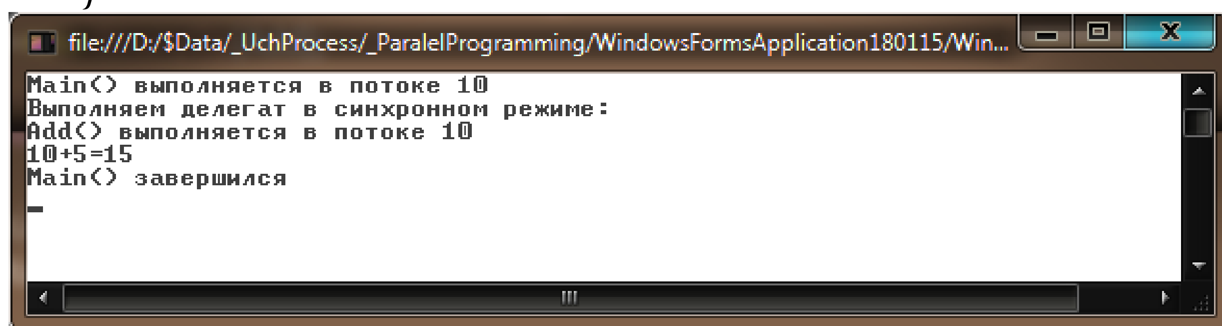

Рис. 2.1.

В представленном фрагменте объявляется тип делегата *BinaryOp* для неявного вызова метода *Add()*. Создание объекта делегата и запуск на выполнение осуществляется в методе *button_Click()*. Вставленные в методы действия вывода информационных сообщений на консоль позволяют наблюдать синхронный режим исполнения делегата.

Из скриншота работы программы следует:

1) выполнение *button_Click()* и *Add()* происходит в одном программном потоке;
2) работа метода *button_Click()* завершится только после завершения выполнения метода *Add()*, запущенного опосредовано через делегат *bop*.

При исполнении метода *Add()* фактически происходит два явления: первое – это запуск метода на исполнение и собственно исполнение метода. Суть синхронного выполнения делегата состоит в том, что выполнение его обработчика строго синхронизировано с методом, из которого запускается делегат (метод *button_Click()*). И вызывающий метод после запуска делегата *bop(10, 5)* будет ждать завершения выполнения его метода обработчика *Add()*.
Таким образом, выполнение делегата в синхронном режиме происходит точно так, как и при явном вызове метода. Отличие при использовании делегата состоит только в неявном вызове метода.

То что вызывающий метод ожидает завершения вызванного делегата является недостатком, если одновременно выполнение обработчика с вызывающим методом допустимо. Так в нашем случае из-за добавленной задержки ожидание составит 5 с, а в это время можно было бы выполнить какие-либо другие действия.

2.2. Асинхронный режим

Кроме синхронного режима возможен и асинхронный режим выполнения делегата. Для того чтобы воспользоваться этим режимом необходимо знать следующее.

2.2.1. Класс делегата

При объявлении типа делегата, например:

```
public delegate int BinaryOp(int x, int y);
```

системой фактически создается ненаследуемый класс делегата, в составе которого имеются следующие методы:

```
public sealed class BinaryOp : System.MultiCastDelegate
{
	public BinaryOp(object target, uint functionAddress);
	public int Invoke(int x, int y);
	public IAsyncResult BeginInvoke(int x, int y, AsyncCallBack cb,
									object state);
	public int EndInvoke(IAsyncResult result);
}
```

В этом классе кроме конструктора, метод *Invoke()* используется для организации выполнения делегата в синхронном режиме, а *BeginInvoke()* и *EndInvoke()* – для асинхронного. Когда мы вызываем делегат, например, *bop(10, 5,)* он исполняется в синхронном режиме методом *Invoke()*, который явно нам не вызывается.

Метод *BeginIvoke( )* запускает делегат и возвращет управление в вызвавший его метод, а значит, последний не простаивает.

В методе *BeginIvoke( )* первые два параметра – это те же параметры, что и в методе *Add()* – параметры делегата. Но после параметров делегата присутствуют еще два параметра:

cb – указатель на функцию обратного вызова (делегат), вызываемую по завершении вызванного на исполнение делегата;

state – ссылка на объект, передаваемый в функцию *cb.*

Метод *BeginInvoke()* возвращает объект типа *IAsyncResult*, который описывает состояние выполнения делегата.

2.2.2. Интерфейс IAsyncResult

IAsyncResult – это интерфейс для представления состояния асинхронной операции, включающий свойства, определенные в табл.2.1.

Таблица 2.1 – Свойства интерфейса *IAsyncResult*

Свойство	Объявление	Описание
AsyncState	*Object AsyncState {get;}*	Возвращает определенный пользователем объект, передаваемый в *BeginInvoke()* при её вызове
Async-WaitHandle	*WaitHandle AsyncWaitHandle {get;}*	Возвращает объект *WaitHandle*, используемый для контроля выполнения асинхронной операции
Completed-Synchronously	*bool CompletedSyn-chronously {get;}*	Возвращает значение, показывающее, синхронно ли закончилась асинхронная операция
IsCompleted	*bool IsCompleted {get;}*	Возвращает значение, показывающее, выполнена ли асинхронная операция

2.2.3. Пример запуска делегата в асинхронном режиме

Запуск делегата в асинхронном режиме позволяет вызывающему процессу не находиться в простое, а выполнять что-то полезное. Но основному процессу также важно и контролировать завершение вызванного им делегата, что реализуется через объект *IAsyncResult*, возвращаемый методом *BeginInvoke( ).*

Для получения результата из запущенного на выполнение в асинхронном режиме делегата необходимо вызвать метод *тип*

EndInvoke( ). Тип возвращаемого значения *тип* совпадает с типом метода делегата.

Вот пример запуска делегата в асинхронном режиме с опросом состояния завершения его выполнения:

```
private void button_Click(object sender, EventArgs e)
    {
     Console.WriteLine("Main() выполняется в потоке {0}  ",
                    Thread.CurrentThread.ManagedThreadId);

       BinaryOp bop=new BinaryOp(Add);
       Console.WriteLine("Выполняем делегат в асинхронном
                                              режиме:");

       IAsyncResult ifatr = bop.BeginInvoke(10, 5, null, null);

       while (!ifatr.IsCompleted)
       {
         Console.WriteLine("ifatr.IsCompleted {0}",
                          ifatr.IsCompleted);
         Thread.Sleep(1500);
         Console.WriteLine("Main() выполняет что-то полезное в
                              ожидании завершения делегата");
       }

       int result = bop.EndInvoke(ifatr);

       Console.WriteLine("10+5={0}", result);
    }
```

После запуска делегата на выполнение основной процесс запускает цикл, который длится, пока вызванный делегат не завершит обработку. В теле этого цикла выполняются другие полезные действия основного процесса. Когда делегат завершит обработку, основной процесс выходит из цикла ожидания и выдаёт полученный результат.

Необходимо отметить, что в асинхронном режиме запуска делегата он выполняется в потоке, отличном от основного (см. рис.2.2).

```
file:///D:/$Data/_UchProcess/_ParalelProgramming/WindowsFormsApplication180115/WindowsFo...
Main() выполняется в потоке 8
Выполняем делегат в асинхронном режиме:
ifatr.IsCompleted False
Add() выполняется в потоке 9
Main() выполняет, что-то полезное в ожидании завершения делегата
ifatr.IsCompleted False
Main() выполняет, что-то полезное в ожидании завершения делегата
ifatr.IsCompleted False
Main() выполняет, что-то полезное в ожидании завершения делегата
ifatr.IsCompleted False
Main() выполняет, что-то полезное в ожидании завершения делегата
10+5=15
```

Рис. 2.2.

2.3. Способы синхронизации делегата с вызывающим потоком

Поскольку основной поток не знает, в какой момент завершится запущенный в асинхронном режиме делегат, возникает проблема синхронизации потоков. Существуют разные способы её решения. Один из способов – это использование свойства *IsCompleted* объекта типа *IAsyncResult,* состояние true которого сигнализирует о завершении асинхронной операции. Этот способ рассмотрен ранее. Кроме этого способа возможны и следующие.

2.3.1. Использование объекта WaitHandle типа IAsyncResult

В составе объекта *IAsyncResult* имеется свойство *AsyncWaitHandle* типа *WaitHandle* пространства *System.Threading,* который предоставляет больше возможностей для контроля асинхронной операции. В частности, в классе *WaitHandle* определен перегруженный метод *WaitOne(),* который используется для блокировки текущего потока до получения сигнала синхронизации. Одна из версий этого метода имеет прототип:

public virtual bool WaitOne(int millisecondsTimeout).

Метод позволяет заблокировать текущий поток до получения объектом *WaitHandle* сигнала синхронизации. Параметр *millisecondsTimeout* определяет период времени ожидания сигнала синхронизации в миллисекундах. Если по истечении этого времени после начала работы метода *WaitOne( )* не получен сигнал, то метод завершает свою работу с результатом *false*. Если сигнал получен, то

возвращаемое значение *true*. Если параметр имеет значение –1 (минус 1), то время ожидания не ограничено.

Далее приводится иллюстрация использования метода на рассмотренном ранее примере.

```
private void button5_Click(object sender, EventArgs e)
    {
       Console.WriteLine("Main() выполняется в потоке {0}  ",
                        Thread.CurrentThread.ManagedThreadId);

       BinaryOp bop = new BinaryOp(Add);
       Console.WriteLine("Выполняем делегат в асинхронном
                                              режиме:");
       IAsyncResult ifatr = bop.BeginInvoke(10, 5, null, null);

       while (ifatr.AsyncWaitHandle.WaitOne(1500))
       {
          Console.WriteLine("ifatr.IsCompleted {0}",
                          ifatr.IsCompleted);
          Thread.Sleep(1500);
          Console.WriteLine("Main() выполняет что-то полезное в
                               ожидании завершения делегата");
       }

       int result = bop.EndInvoke(ifatr);

       Console.WriteLine("10+5={0}", result);
    }
```

На рис.2.3 представлен результат работы примера программы. Отметим, что поскольку в данном примере длительность исполнения асинхронной операции (5000 мс) превышает в несколько раз период времени ожидания сигнала (1500 мс), то тело цикла ожидания выполнится несколько раз. Если не существует никаких действий, которые можно было бы выполнить в цикле ожидания, то от цикла можно отказаться совсем, заменив его на строку вида:
ifatr.AsyncWaitHandle.WaitOne(-1);

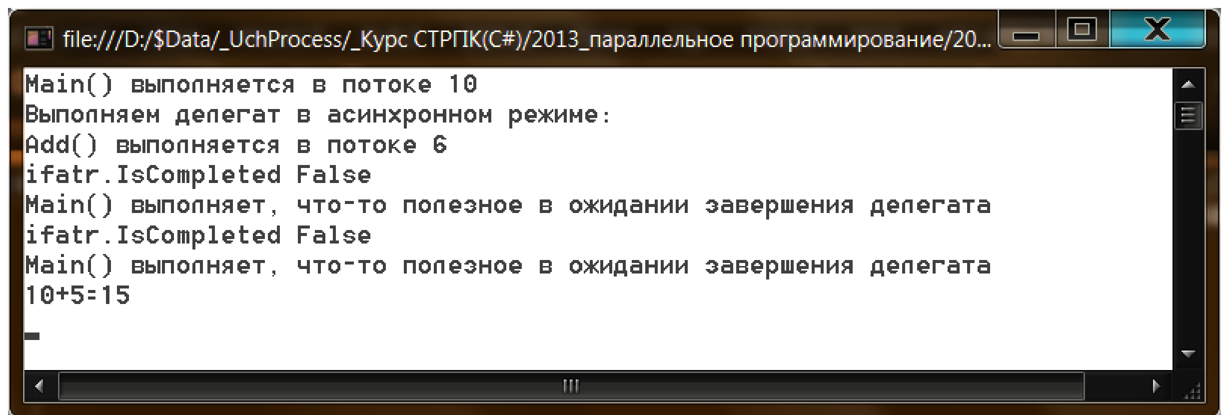

Рис. 2.3.

2.3.2. Использование функции обратного вызова

Как было сказано раньше при вызове функции *BeginInvoke()* можно указать параметр *cb,* определявший функцию обратного вызова, которая должна иметь прототип:

void MyAsyncCallBack(IAsyncResult v).

Эта функция будет вызвана по завершению исполнения делегата. Её можно использовать для выполнения, каких угодно целей. Например, в эту функцию можно перенести завершающие действия после выполнения асинхронной операции. Также, используя эту функцию можно самостоятельно организовать синхронизацию основного потока и асинхронной операции.

Для иллюстрации в состав класса, организующего основной поток введём новые элементы:

```
    private  static bool isDone=false;


    private static void AddComplete(IAsyncResult ifatr)
    {
       Console.WriteLine("AddComplete() выполняется в потоке {0}",
                              Thread.CurrentThread.ManagedThreadId);
       Console.WriteLine("Add() завершено!");

       AsyncResult ar = (AsyncResult)ifatr;

       BinaryOp bop = (BinaryOp)ar.AsyncDelegate;
```

```
        string data = (string)ifatr.AsyncState;

        Console.WriteLine("были переданы " + data);
        int result = bop.EndInvoke(ifatr);
        Console.WriteLine("10+5={0}", result);

        isDone = true;
    }
```

А метод, вызывающий асинхронную операцию, изменим так:

```
    private void button_Click(object sender, EventArgs e)
    {
        Console.WriteLine("Main() выполняется в потоке {0}  ",
                          Thread.CurrentThread.ManagedThreadId);
        BinaryOp bop = new BinaryOp(Add);
        Console.WriteLine("Выполняем делегат в асинхронном
                                                    режиме:");
        IAsyncResult ifatr =
              bop.BeginInvoke(10, 5, new AsyncCallback(AddComplete),
                    "Любые данные для функции обратного вызова");
        isDone = false;

        while (!isDone)
        {
            Thread.Sleep(1500);
            Console.WriteLine("Main() выполняет что-то полезное в
ожидании завершения делегата");
        }
    }
```

Результат работы представлен на рис.2.4. В данном примере результат асинхронной операции на консоль выводится функцией *AddComplete( ).* Эта функция вызывается по завершении выполнения делегата. При вызове в неё передаётся объект синхронизации *ifatr,* связанный с делегатом. Именно через этот объект функция *AddComplete( )* получает доступ к фактическому параметру *"Любые данные для функции обратного вызова"* (свойство *ifatr.AsyncState*) и к результату выполнения асинхронной операции. Для синхрониза-

ции основного потока и асинхронной операции используется флаг состояния её завершения – *isDone,* устанавливаемый в методе *AddComplete( ).* По значению *true* этого флага основной поток узнает о завершении асинхронной операции. Если основной поток не должен реагировать на состояние завершения асинхронной операции, цикл ожидания завершения можно исключить.

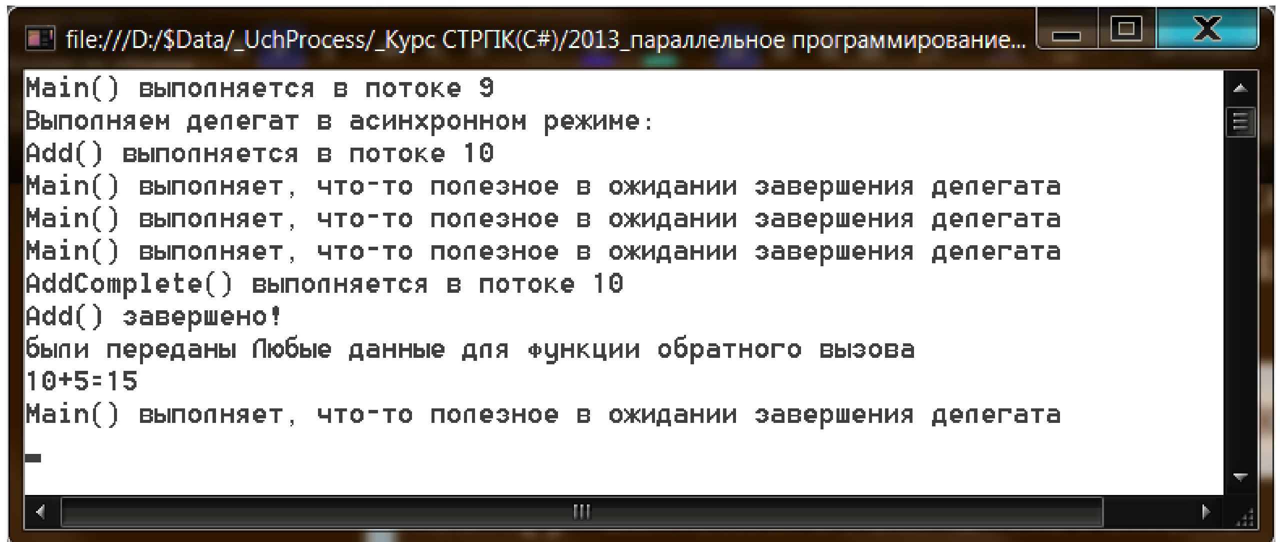

Рис. 2.4.

Для синхронизации потоков при выполнении асинхронной операции можно определить событие самостоятельно, как показано в разделе 1.2. Событие должно выбрасываться в функции обратного вызова, а обработчик этого события должен устанавливать флаг состояния завершения асинхронной операции и выводить её результат.

3. Задание для лабораторной работы

Тема: Использование делегатов и событий для разработки приложений.

Цель работы: Освоение технологии разработки приложений на языке C# с использованием делегатов и событий.

Для выполнения работ могут быть использованы системы программирования *MS Visual Studio .Net 2013, 2015*.

Задание:

1) Изучить типы делегата и события.
2) Освоить способы исполнения делегатов в синхронном и асинхронном режимах.
3) Освоить принципы разработки приложения с использованием событий и делегатов.
4) Освоить принципы построения параллельных приложений на основе делегатов.
5) Разработать приложения с использованием делегатов и событий для решения предложенных задач.

Порядок выполнения работы:

1. Пункты задания 1 – 4 изучить по соответствующим разделам данных методических указаний.
2. Пункт 5 задания отработать на примере решения следующих ниже задач.

Задача 1

Разработать приложение для вычисления интеграла выбираемой функции из заданного списка функций выбираемым методом из заданного списка методов. Для реализации использовать два класса. Первый класс – *Program*, организует меню приложения, определяет список функций, вводит исходные данные, выводит полученные результаты. Второй класс – *Integral*, реализует методы

для вычисления интеграла (не менее трёх). Для указания функции и метода при вычислении интеграла использовать делегаты в синхронном режиме.

Задача 2

Для функций выбираемых из перечня функций (не менее пяти) на заданном интервале аргументов *[xb, xe]* с шагом *dx (xb, xe, dx* –вводятся) распечатать таблицу значений функций. Для реализации использовать два класса. Первый класс – *Program*, организует меню приложения, определяет список функций и вводит исходные данные для печати таблицы:

- ширина поля таблицы в символах;
- выравнивание значений в таблице (*по левому краю*, *по правому краю*);
- количество знаков после десятичной точки;
- режим вывода (*чистовой* – каждая строка таблицы выводится в отформатированном виде с представлением значений в формате с фиксированной точкой, *черновой* – каждая строка выводится в неотформатированном виде с представлением значений в формате с плавающей точкой);
- разделитель строк (*есть* – строки разделяются горизонтальной линией, *нет* – печатаются без разделителя);
- разделитель полей (*есть* – печатается, *нет* – не печатается);
- символ разделителя полей (по умолчанию символ !);
- количество пробелов после поля и перед полем (по умолчанию 1);
- текст, который выводится в случае неопределённого значения функции (по умолчанию – «неопр.»).

Второй класс – *Table*, для заданного перечня функций, определяемых делегатами, и для заданных параметров таблицы, организует построение таблицы, но сам не выводит данные на консоль. В момент, когда очередная строка таблицы сформирована и готова к выводу на консоль, выбрасывается событие *NextString*. Обработчики этого события определяются в классе *Program*, они выводят очередную строку таблицы в черновом и чистовом режиме, и вывод разделительной линии.

Задача 3

Для задачи 1 в классе *Integral* определить метод, который организует вычисление интеграла в асинхронном режиме. Отработать разные способы синхронизации потоков с использованием:

- флага *IsCompleted* объекта *IAsyncResult;*
- объекта *WaitHandle* типа *IAsyncResult;*
- функции обратного вызова как параметра *BeginInvoke()* с установкой флага завершения вычисления в классе *Program;*
- функции обратного вызова как параметра *BeginInvoke()* с выбросом события завершения вычисления и обработчиком этого события в классе *Program.*

Задача 4

Разработать приложение, моделирующее работу конвейера обработки данных, на вход которого поступают объекты данных из входного файла. После обработки на конвейере объекты результаты записываются в результирующий файл, т.е. каждому исходному объекту данных из входного файла соответствует объект данных результат в результирующем файле.

Конвейер обработки данных включает *K* ступеней, на каждой из которых выполняется одна операция обработки *OPi (i=1,K)* над каждым объектом данных из входного файла. Для моделирования конвейера использовать такие правила:

1) Объект данных из входного файла поступает на ступень 1.
2) Результат обработки объекта данных ступени *i* является входным объектом данных для ступени *i+1*, если $i+1 \leq K$, или записывается в результирующий файл в противном случае.
3) На каждой ступени конвейера может выполняться операция этой ступени над одним объектом.
4) Порядок обработки объектов данных на ступенях конвейера совпадает с исходным порядком объектов данных в файле и не может быть нарушен.
5) Каждая ступень конвейера имеет свою длительность выполнения операции.
6) Если ступень *i* завершила обработку объекта данных, а ступень *i+1* не готова (не свободна), то объект данных – результат

ступени *i* продолжает занимать ступень, пока не перейдёт на ступень $i+1$.

Каждую ступень конвейера моделировать делегатом, исполняемым в асинхронном режиме, так чтоб конвейер работал в живом времени. В процессе моделирования на консоль выводить информацию о загрузке каждой ступени конвейера, об освобождении каждой ступени с указанием момента времени (отсчёт ведётся от момента первой загрузки ступени 1).

По результатам моделирования построить график Ганта загрузки конвейера, на котором выделять время обработки и простоя.

Содержание

Введение..3

1. Делегаты и события 4

1.1. Делегаты...4

1.2. События 9

2. Режимы выполнения делегатов 24

2.1. Синхронный режим...24

2.2. Асинхронный режим..25

2.2.1. Класс делегата 26

2.2.2. Интерфейс IAsyncResult 26

2.2.3. Пример запуска делегата в асинхронном режиме.....................27

2.3. Способы синхронизации делегата с вызывающим потоком........29

2.3.1. Использование объекта WaitHandle типа IAsyncResult 29

2.3.2. Использование функции обратного вызова 31

3. Задание для лабораторной работы 34

Учет использования методических указаний

№	Ф.И.О. студента	Дата	Роспись

Навчальне видання

Методичні вказівки
до лабораторних робіт «Делегати та події в мові C#»
з дисципліни «Технологіїї програмування»
для студентів спеціальностей
122 – Компютерні науки та інформаційні технології,
124 – Системний аналіз, в тому числі для іноземних студентів

Російською мовою

Укладачі:
МАЛИХ Олег Миколайович
КОЖИН Юрій Миколайович
ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск О. С. Куценко

Роботу до друку рекомендував О. В. Горєлий

Редактор О. І. Шпільова

План 2017 , поз.112

Підп. до друку. Формат 60х84 1/16. Папір офсетний.
Друк – ризографія. Гарнітура Таймс. Ум. друк. арк. 1,7.
Обл.-вид. арк. 4,7 Наклад 100 прим Зам. № 75 Ціна договірна.

Видавничий центр НТУ "ХПІ",

61002, м.Харків, вул. Кирпичова, 2

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

Друкарня НТУ "ХПІ" . 61002, м.Харків, вул. Кирпичова, 2