Делегаты и события

Лекция №10

Понятие делегата

Делегат – это вид класса, объекты которого могут ссылаться на методы.

Его базовым классом является класс **System.Delegate.**

Наследовать от делегата нельзя.

Делегат может вызвать метод, на который он ссылается.

Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, т.е. метод, вызываемый делегатом, определяется не в период компиляции программы, а во время ее работы.

Делегаты предоставляют следующие возможности:

- динамическое определение вызываемого метода (не при компиляции, а во время выполнения программы);
- обеспечение связи вида «источник - наблюдатель» между объектами;
- создание универсальных методов, в которые можно передавать другие методы;
- поддержка механизма обратных вызовов.

Описание делегата

[спецификатор] delegate <тип> имя_делегата([параметры]);

Спецификаторы: new, protected, internal, private, public.

Например:

public delegate double func(double x, y) ;

Описание делегата можно размещать как непосредственно в пространстве имен, так и внутри класса.

Описание делегата задает сигнатуру методов, которые могут быть вызваны с помощью этого делегата.

Использование делегата

Делегат может вызывать либо метод экземпляра класса, связанный с объектом, либо статический метод, связанный с классом.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно. Методы вызываются последовательно в том порядке, в котором они были добавлены в делегат.

Перед использованием делегат нужно создать и инициализировать методом, на который будет ссылаться делегат:

<Тип делегата> < имя> = new <тип делегата>(<имя метода>);

Например, если в классе описан делегат

public delegate double func(double x, y) ;

а **my_func** – это метод с такой же сигнатурой как в описании делегата, то создание экземпляра делегата может быть таким:

func F = new func(my_func);

или

func F;

F = new func(my_func);

Вызов метода с помощью делегата осуществляется так:

<имя экземпляра делегата>(<список аргументов>);

Количество и тип аргументов в списке совпадают с количеством и типом параметров в описании делегата.

Например, **F(3.5,2);**

Попытка вызова делегата, для которого не указан метод на который этот делегат будет ссылаться, приведет к генерации исключения **System.NullReferenceException.**

Рассмотрим пример, демонстрирующий возможность динамически определять какой из методов в тот или иной момент выполнения программы следует вызвать.

Пусть требуется вычислить значение одной из функций по выбору пользователя:

$$f1(x, y) = \sin(x) + \cos(y)$$

$$f2(x, y) = \sin(x + y)$$

А также получить таблицу значений функции sin(t) для t принимающего значения от x до y.

```
namespace ConsoleApplication1
{
    public   delegate   double    func(double x, double y);
     class function
        {
           public  static  double    f1(double x, double y)
           { return Math.Sin(x) + Math.Cos(y); }
           public static  double    f2(double x, double y)
                { return Math.Sin(x + y); }
         }
```

```
class   Program
   {
      public   static  double    tab(double xn, double xk)
      {
         double y=0;
         for (double x = xn; x <= xk; x = x + 0.1)
           { y = Math.Sin(x); Console.WriteLine(x+"   "+y); }
         return y;
       }
```

```
static void Main(string[] args)
        {
            Console.WriteLine("Введите x и y");
            double x = Convert.ToDouble(Console.ReadLine());
            double y = Convert.ToDouble(Console.ReadLine());

        Console.WriteLine("Задайте номер функции для расчета");
             int n = Convert.ToInt32(Console.ReadLine());

          func     F= null;
```

```
if(n = = 1)  F= new    func(function.f1);
  else if ( n= = 2)   F = new    func(function.f2);

double   f =  F(x,y);
Console.WriteLine("x =" +x+"  y="+y+"  f="+f);

   F = new     func(tab);
    F(x, y);
    Console.ReadKey();
          }
      }
   }
```

Если методы f1 и f2 будут нестатическими, то в программе понадобятся следующие изменения:

```
function FF = new function( );

func F = null;
if(n= = 1) F=new func(FF.f1);
 else if (n= = 2) F=new func(FF.f2);
```

Многоадресатная передача

Многоадресатная передача – это способность создавать список вызовов методов, которые будут автоматически вызываться при вызове делегата.

Если делегат хранит ссылки на несколько методов, то при вызове делегата эти методы вызываются последовательно в том порядке, в котором они были добавлены в делегат.

Для добавления метода в список можно использовать операцию «+=» или отдельно «+» и «=».

Для удаления метода из списка можно использовать операцию «-=» или отдельно «-» и «=».

<Тип делегата> < имя> = new <тип делегата>(<имя метода 1>);

< имя> += new <тип делегата>(<имя метода 2>);

. . .

< имя> += new <тип делегата>(<имя метода n>);

Можно без new:

< имя> += <имя метода 2>;

Каждому методу в списке передается один и тот же набор параметров.

Если параметр методов списка описан с ключевым словом **out** или методы возвращают значение, результатом выполнения делегата будет значение, сформированное последним методом в списке.

Поэтому рекомендуется, чтобы делегат с многоадресатной передачей возвращал тип **void**.

Если требуется передача измененных параметров, их нужно описать с ключевым словом **ref**.

Приведем пример, иллюстрирующие использование делегатов с многоадресатной передачей.

```
public  delegate  void    func(double x, double y);
class function
   {
      public    void    f1(double x, double y)
      { double z = Math.Sin(x) + Math.Cos(y);
  Console.WriteLine("x =" + x + "  y=" + y + "  sin(x)+cos(y)=" + z);
        }
   public   static   void    f2(double x, double y)
          {  double z = Math.Sin(x + y);
          Console.WriteLine("x =" + x + "  y=" + y + "  sin(x+y)=" + z);
          }
   }
```

```
class Program
    {
       public    static void   tab(double xn, double xk)
         {
                  double   y;
                  for (double   x = xn; x <= xk+0.05; x = x + 0.1)
                  { y = Math.Sin(x); Console.WriteLine(x+"   "+y); }

          }
```

```
static void Main(string[ ] args)
    {
       Console.WriteLine("Введите x и y");
       double x = Convert.ToDouble(Console.ReadLine( ));
       double y = Convert.ToDouble(Console.ReadLine( ));

        function   FF = new   function( );
        func    F= new    func(FF.f1);
         F += new    func(function.f2);
         F =F + new   func(tab);

          F(x, y);
```

или так:

```
func  F= new   func(FF.f1);
 F += function.f2;
   F = F+ tab;
   F(x, y);
```

В результате выполнения этого фрагмента программы будут получены следующие результаты:

x =1 y=2 sin(x)+cos(y)=0,425324148260754

x =1 y=2 sin(x+y)=0,141120008059867

1 0,841470984807897

1,1 0,891207360061435

1,2 0,932039085967226

1,3 0,963558185417193

1,4 0,98544972998846

1,5 0,997494986604054

1,6 0,999573603041505

1,7 0,991664810452469

1,8 0,973847630878195

1,9 0,946300087687414

2 0,909297426825681

После выполнения следующих операторов

F - = function.f2; F(x, y);

получим следующий результат:

x =1 y=2 sin(x)+cos(y)=0,425324148260754

1 0,841470984807897

1,1 0,891207360061435

1,2 0,932039085967226

1,3 0,963558185417193

1,4 0,98544972998846

1,5 0,997494986604054

1,6 0,999573603041505

1,7 0,991664810452469

1,8 0,973847630878195

1,9 0,946300087687414

2 0,909297426825681

Следующий пример показывает передачу значений при многоадресной передаче.

```
public delegate void str(ref string s);

class Stroki
  {
    public static void rs(ref string s)
      { s = s.Replace('-', '*'); }

    public static void rs1(ref string s)
      { s = s + "!!!"; }
  }
```

Тогда выполнение следующих операторов

```
string    stroka =  "Ча-ча-ча";

str S = null;

S +=  Stroki.rs;   S +=  Stroki.rs1;

S(ref   stroka);

Console.WriteLine(stroka);
```

приведет к результату:

Ча*ча*ча!!!

Передача делегатов в методы

Так как делегат – это класс, его экземпляры можно передавать в методы в качестве параметра.

Благодаря этому можно создавать *универсальные* методы.

Примером такого универсального метода может служить метод для вывода таблицы значений функции.

```
class Program
  {
     public  delegate  double Func(double x);
```

```
public  static void Tab(double  xn, double  xk, double  dx, Func  F)
   {
Console.WriteLine("╔═══════════════════════════════════════════╗");
Console.WriteLine("║     Аргумент      ║     Функция           ║");
Console.WriteLine("╚═══════════════════╩═══════════════════════╝");
       double x = xn;
         while (x  <  xk + dx / 2)
         { Console.WriteLine("│ {0,12:f2} │ {1,15:f3} │", x, F(x));
           x = x + dx;
         }
Console.WriteLine("└──────────────────┴──────────────────────┘");
       }
```

```
public static double f1(double x)
    { return        Math.Cos(x) + Math.Sin(x); }
public static double f2(double x)
     { return      Math.Cos(x) - Math.Sin(2*x); }

  static void Main(string[ ] args)
         {
            Console.WriteLine("cos(x)+sin(x)");
            Tab( -3, 3, 0.5,  new   Func(f1));
```

```
Console.WriteLine("cos(x)-sin(2x)");
Tab(-2, 2, 0.2, new   Func(f2));

 Console.WriteLine("cos(x)");
 Tab(0, 2, 0.1, new Func(Math.Cos));
          Console.ReadKey( );
      }
   }
```

Можно не создавать экземпляр делегата явным образом с помощью операции new. Он будет создан *автоматически*.

Т.е. в программе можно использовать такие операторы:

Tab(-3, 3, 0.5, f1);

Tab(-2, 2, 0.2, f2);

Tab(0, 2, 0.1, Math.Cos);

Например, наряду с оператором

Func FF = new Func(Math.Sin);

правомерно использование оператора

Func FF = Math.Sin;

Вышесказанное относится к версии 2.0 языка C#.

Если делегат описан внутри какого-то класса, то для использования его в другом классе требуется указывать имя класса, содержащего описание делегата.

Например,

```
class dd {
        public  static  void      z(Program.Func   F,  double x)
              { Console.WriteLine(F(x)); }
          }
```

Тогда в методе Main класса Program можно применять такой оператор:

```
dd.z(f1, 3);
```

В версии 2.0 C# можно создавать *анонимные* методы в качестве аргумента метода, подставляемого вместо параметра-делегата.

Анонимный метод – это фрагмент кода, описываемый в месте использования делегата следующим образом:

delegate(<список параметров>) {<тело метода>}

Например,

Tab(0, 2, 0.1, **delegate(double x) { return x*x+3; }**);

или

```
Func  FFF =  delegate(double x)
                    { if (x > 0) return -1; else  return 1; };
```

Tab(-2, 2, 0.2, FFF);

Обеспечение связи «источник - наблюдатель» между объектами

При использовании множества совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов.

Т.е. при изменении в одном объекте во всех связанных с ним объектах других классов тоже должны произойти необходимые изменения, чтобы не нарушалась согласованность.

Например, если в объекте класса **Факультет** произошла смена названия факультета, на это должны автоматически среагировать все объекты класса **Студент**, обучающиеся на этом факультете.

Для обеспечения такой связи между объектами применяется следующая схема (шаблон проектирования, паттерн), получившая название паттерн «наблюдатель».

Объект-источник при изменении своего состояния посылает уведомление **объектам-наблюдателям**, заинтересованным в информации об этом изменении, и наблюдатели синхронизируют свое состояние с источником.

Таким образом, при изменении состояния одного объекта все зависящие от него объекты извещаются и автоматически обновляются.

Для реализации стратегии «наблюдатель» в классе, объекты которого будут выступать в роли источника, нужно предусмотреть возможность регистрации наблюдателей, чтобы объект-источник «знал», кого нужно оповестить об изменении состояния.

Для этого можно использовать делегат с многоадресатной передачей: при регистрации будет формироваться список вызовов из методов объектов-наблюдателей, которые будут запускаться для реагирования на изменение в объекте-источнике.

Например,

```
public delegate void  Nab(object ob);     // описание делегата,
// объект которого будет содержать ссылки на методы,
//  вызываемые при изменении в источнике
    class Bank
        {   public string name;
            double kurs_dol;
            Nab   Nab_li;  // экземпляр делегата для регистрации
                           //   наблюдателей
```

```
public    void      Registracija(Nab   N) { Nab_li  +=  N; }

public void K_D(double   kd)          // метод, в котором //происходит
                                      интересующее    //наблюдателей
      { kurs_dol = kd;                изменение
        if (Nab_li != null)  Nab_li(this);// оповещение  наблюдателей
      }

  public double   KD { get { return    kurs_dol; } } }
```

```
class Student
    {
        string   Fam;
        double   stip,  stip_dol;

         public Student(string   fm, double   st)
                  { Fam = fm;   stip =  st;  stip_dol = st / 2200; }

public void vivod( )
  { Console.WriteLine("Я - " + Fam + ",  моя стипендия: " + stip_dol + "$"); }
```

```
 public   void S_D(object    ob) // реакция на изменение в
       {                                  // источнике
          stip_dol = stip/((Bank)ob).KD;
       }
}

 class Program
    {
       static void Main(string[  ] args)
       {
          Bank B = new   Bank( );      B.name = "Беларусбанк";
```

Создание объекта источника

```
Student   S1 = new  Student("Тяпкин", 150000);
Student   S2 = new  Student("Ляпкин", 150000);
Student   S3 = new  Student("Пупкин", 150000);
```

потенциальные наблюдатели

```
B.Registracija(S1.S_D);
B.Registracija(S2.S_D);
```

регистрация наблюдателей

```
B.K_D(2500);      // изменение состояния источника

S1.vivod( );   S2.vivod( );      S3.vivod( );
            Console.ReadKey();
        }
    }
```

Результат выполнения программы:

Я – Тяпкин, моя стипендия: 60$

Я – Ляпкин, моя стипендия: 60$

Я – Пупкин, моя стипендия: 68,1818181818182$

Для того чтобы наблюдатель «знал», от какого источника пришло уведомление, делегат объявлен с параметром типа **object**.

Через этот параметр в вызывающий метод передается ссылка на объект-источник.

Таким образом, наблюдатели получают доступ к элементам источника.

Операции с делегатами.

Для делегатов можно использовать операции **равно ==** и **не равно !=.**

Делегаты равны, если они либо оба не содержат ссылок на методы, либо их списки вызовов совпадают, т.е. эти делегаты содержат ссылки на одни и те же методы в одном и том же порядке.

Сравнивать можно только те делегаты одного типа.

Например, пусть описан делегат:

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

Тогда в результате выполнения операторов

```
Func   F1 = Math.Sin;   F1 += Math.Cos;
Func   F2 = Math.Cos;    F2 += Math.Sin;
Console.WriteLine("F1 = F2  :  " + (F1 == F2));
```

получим **F1 = F2 : False**

А после выполнения

```
Func F3 = null;
Func F4 = null;
Console.WriteLine("F4 = F4  :  " + (F3 == F4));
```

получим **F4 = F4 : True**

Для однотипных делегатов можно выполнять операции присваивания, **+=** и **-=**

Результатом этих операций будет новый экземпляр, так как делегат является неизменяемым типом данных.

Результатом операции присваивания будет экземпляр, содержащий список вызовов, идентичный списку вызываемых методов объекта в правой части оператора.

Например, после выполнения оператора

```
Func F = F1;
```

делегат F будет содержать ссылки на методы Math.Sin и Math.Cos.

Результатом операции += будет делегат, список вызовов которого содержит список вызовов делегата в левой части оператора с добавленным к нему списком делегата в правой части.

Например, пусть описан делегат

```
public delegate void Func1(double x);
```

А в классе **Program** описаны методы:

```
public static void V1(double x)
                                                    {
Console.WriteLine("cos="+Math.Cos(x));
        }

public static void   V2(double x)
       { Console.WriteLine("sin="+Math.Sin(x)); }

 public static void   V3(double x)
        { Console.WriteLine("2x = " + 2*x); }
```

Тогда после выполнения операторов

```
Func1    FF1 = V1;    FF1 += V2;
Func1    FF2 = V3;
 FF2  +=  FF1;
    FF2(3);
```

будет получен следующий результат:

2x = 6

cos= -0,989992496600445

sin= 0,141120008059867

Результатом операции -= будет делегат, список вызовов которого содержит список вызовов делегата в левой части оператора из которого удален список делегата в правой части.

Например,

FF2 -= FF1; FF2(3);

Результат: **2x = 6**

Если список вызовов делегата в правой части оператора не совпадает с частью списка делегата в левой части, список вызовов остается неизмененным.

Например,

```
Func1   FF3 = delegate(double x)
                    { Console.WriteLine("x=" + x); };
          FF2 -= FF3;   FF2(3);
```

Результат:

2x = 6

cos=-0,989992496600445

sin=0,141120008059867

После выполнения

```
Func1  FF3 = V2;  FF3 += V1;
          FF2 -= FF3;  FF2(3);
```

результат тот же.

С делегатами одного и того же типа можно выполнять операции сложения и вычитания:

FF3 = FF2 + FF1; FF3= FF2-FF1;

Пример (для лаб. работы).

Подготовить текстовый файл, содержащий информацию о подразделениях фирмы: название отдела, номер телефона, размер премии в процентах к заработной плате.

Подготовить текстовый файл, содержащий информацию о сотрудниках фирмы: фамилия, название отдела, стаж, зарплата в $.

Разделителем служит символ «;»

Например, baza1.txt:

Плановый отдел;48-35-80;10

Отдел кадров;48-15-35;15

Бухгалтерия;48-05-55;5

baza2.txt:

Плюшкин И.И.;Бухгалтерия;15;200

Коклюшкин В.В.;Плановый отдел;20;300

Финтифлюшкин С.А.;Бухгалтерия;12;150

Замухрышкин Я.Я.;Отдел кадров;35;350

Телевышкин Ы.Ы.;Плановый отдел;10;200

Открывашкин Ъ.Ъ.;Отдел кадров;20;260

Разработать программу, которая выполняет следующие действия:

- Считывает информацию из первого файла в массив объектов класса **Отдел**.

Класс **Отдел** должен содержать следующие элементы: поля для хранения названия отдела, телефона и размера премии в процентах, свойства для доступа к полям, поля и методы для реализации шаблона «наблюдатель» (см. ниже),.

- Считывает информацию из второго файла в массив объектов класса **Сотрудник**.

Класс **Сотрудник** должен содержать следующие элементы: поля для хранения фамилии, названия отдела, рабочего телефона, зарплаты, премии в $, свойства для доступа к полям, методы для реализации шаблона «наблюдатель» (см. ниже).

- Выводит информацию о сотрудниках по отделам.

- Реализует схему «наблюдатель» без использования событий, в которой при изменении телефона отдела автоматически изменяется номер телефона для всех сотрудников этого отдела. Для этого каждый сотрудник должен быть «зарегистрирован» в своем подразделении. Отдел, номер которого меняется, выбирается пользователем.

- Реализует шаблон «наблюдатель», в котором при изменении размера премии в процентах для сотрудников, стаж которых больше 12 лет автоматически изменяется размер премии. Все сотрудники, стаж которых больше 12 лет должны быть «зарегистрированы» в своем отделе. Премия будет увеличиваться на 5%.

Реализация:

```
public  delegate  void Nab(object  ob);

class   Otdel
    {
```

```
string   name, telefone;
int    premija;

  Nab     Nab_li_prm,  Nab_li_tlfn;

  public  Otdel(string   nm, string   tlfn, int   prm)
          { name = nm;   telefone = tlfn;   premija = prm; }

public   void    Registracija_prm(Nab  N)  { Nab_li_prm += N; }

 public    void     Registracija_tlfn(Nab   N) { Nab_li_tlfn += N; }
```

```
public void   Prem(int    procent)
      {
         premija = procent;
         if (Nab_li_prm != null)    Nab_li_prm(this);
      }

 public   void    Tel(string    tlfn)
       {
          telefone = tlfn;
          if (Nab_li_tlfn != null)  Nab_li_tlfn(this);
       }
```

```
public string Name
        { get { return name; } }

public int   Premija
        { get { return   premija; } }

  public   string   Telefone
          { get { return telefone; } }

public   void   vivod( )
        { Console.WriteLine(name+" "+telefone+"  "+premija); }
    }
```

```
class   Sotrudnic  :  IComparable
    {   string    fam, naz_otdel, rab_tel;
         int        staz;
        double   zarplata, premia;

public   Sotrudnic(string    fm, string    nazotd, int   stz, double   zpl)
       { fam = fm;   naz_otdel = nazotd;  staz = stz;  zarplata = zpl;
       }
public   void     vivod( )
      { Console.WriteLine(fam + " " + rab_tel + "   " + staz+
                                        " "+(zarplata+premia)); }
```

```
public    void    PR(object     ob)
        { premia = ((Otdel)ob).Premija * zarplata / 100; }


public     void    R_T(object  ob)
        { rab_tel = ((Otdel)ob).Telefone; }


 public string   Naz_o { get { return naz_otdel; } }
 public int        Staz     { get { return staz; } }
```

```
public  int  CompareTo(object obj)
        {
            Sotrudnic   std = obj  as  Sotrudnic;
            return   naz_otdel.CompareTo(std.naz_otdel);
        }
    }

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

```
StreamReader    f1 = new    StreamReader("baza1.txt");
        string   s = f1.ReadLine(); int j = 0;
        while (s != null)
        {
          s = f1.ReadLine();
          j++;
        }
       f1.Close();
```

```
Otdel[ ]   ot = new  Otdel[j];

f1 = new    StreamReader("baza1.txt");
          s =  f1.ReadLine( ); j = 0;

        while (s != null)
                {
                   string[ ]   ss = s.Split(';');

            ot[j] = new   Otdel(ss[0],ss[1],Convert.ToInt32(ss[2]));

                    s = f1.ReadLine();
                              j++;
                }
```

```
f1.Close();
foreach  (Otdel  otd  in  ot)   otd.vivod( );

 StreamReader f2 = new StreamReader("baza2.txt");
          s = f2.ReadLine();  j = 0;
          while (s != null)
          {
              s = f2.ReadLine();
              j++;
          }
          f2.Close();
```

```
Sotrudnic[ ]  st = new Sotrudnic[j];

 f2 =  new     StreamReader("baza2.txt");
   s = f2.ReadLine();     j = 0;

  while (s != null)
            {
               string[ ] ss = s.Split(';');

st[j] = new     Sotrudnic(ss[0], ss[1], Convert.ToInt32(ss[2]),
                                                  Convert.ToDouble(ss[3]));

                  int i = 0;
```

```
while (i < ot.Length)
   { if (st[j].Naz_o = = ot[i].Name)
         { ot[i].Registracija_tlfn(st[j].R_T);
            if   (st[j].Staz > 12)   ot[i].Registracija_prm(st[j].PR);
              i = ot.Length;
          }
        i++;
     }
```

```
  s = f2.ReadLine();
                j++;
  }
  f2.Close();

foreach (Otdel   otd   in    ot)
          { otd.Tel(otd.Telefone);   otd.Prem(otd.Premija); }

   Array.Sort(st);

    Console.WriteLine(st[0].Naz_o);     st[0].vivod();
```

```
for (int i = 1; i < st.Length; i++)

 { if (st[i].Naz_o != st[i - 1].Naz_o)
                                    Console.WriteLine(st[i].Naz_o);
     st[i].vivod();
  }

Console.WriteLine(" В каком отделе изменяется телефон?");
          string s_o = Console.ReadLine( );
          Console.WriteLine(" Введите номер  телефона?");
          string t_o = Console.ReadLine( );
```

```
  foreach (Otdel  otd  in   ot)
          {if  (otd.Name == s_o)   otd.Tel(t_o);  }


Console.Clear();
          Console.WriteLine(st[0].Naz_o);  st[0].vivod();
          for (int i = 1; i < st.Length; i++)
          {
             if (st[i].Naz_o != st[i - 1].Naz_o)
                                Console.WriteLine(st[i].Naz_o);
             st[i].vivod();
          }
```

```
Console.WriteLine("Премия увеличивается на 5%");
        foreach (Otdel otd in ot)
                                { otd.Prem(otd.Premija+5); }
        foreach (Sotrudnic  std  in  st)   std.vivod();


    Console.ReadKey();
          }
     }
```

События

Событие – это элемент класса, позволяющий автоматически уведомлять объекты других классов об изменении своего состояния.

События, таким образом, позволяют реализовать шаблон «наблюдатель».

События работают следующим образом.

Объект-наблюдатель, которому необходима информация о некотором событии, регистрирует метод-обработчик для этого события в объекте-источнике события.

Когда ожидаемое событие происходит (посылается уведомление), вызываются все зарегистрированные обработчики.

Методы-обработчики событий вызываются с помощью делегатов.

Простая форма описания события в классе:

[спецификаторы] event <тип> <имя события>;

Спецификаторы: public, protected, internal, private, static, virtual, sealed, override, abstract.

Тип события – это тип делегата, на основе которого построено событие.

Чтобы использовать в классе событие, нужно

- Описать делегат, задающий сигнатуру обработчиков события.

- Описать в классе событие соответствующего типа.

- Создать один или несколько методов, инициирующих событие.

Внутри этого метода должно быть обращение к методам-обработчикам посредством события.

Так как событие – это фактически экземпляр делегата, то вызов методов-обработчиков с помощью события осуществляется так:

<имя события>(<список аргументов>);

Поскольку события обычно предназначены для многоадресатной передачи, они должны возвращать значение типа **void.**

Для регистрации метода-обработчика (добавление к событию) используется оператор вида:

< имя источника>.<имя события>

+= new <тип делегата>(<имя метода>);

или

< имя источника>.<имя события> += <имя метода>;

Например:

```
public   delegate    void     Nab(object    ob);

 class Bank
     {   public string name;
         double kurs_dol;

           public   event     Nab     Nab_za_kurs_d;
public  void  K_D(double kd)
        {   kurs_dol = kd;
            if (Nab_za_kurs_d != null)    Nab_za_kurs_d(this);
        }
```

```
public double KD { get { return kurs_dol; } }
    }

  class Student
     {
        string Fam;
        double stip,stip_dol;

         public Student(string fm, double st)
                  { Fam = fm; stip = st; stip_dol = st / 2200; }
```

```
public void   S_D(object ob)
      {
         stip_dol = stip/((Bank)ob).KD;
      }

public void vivod( )
       { Console.WriteLine("Я - " + Fam + "  моя стипендия: " +
                                          stip_dol + "$"); }
  }
```

```
Bank B = new Bank(); B.name = "Беларусбанк";
        Student S1 = new Student("Тяпкин", 150000);
        Student S2 = new Student("Ляпкин", 150000);
        Student S3 = new Student("Пупкин", 150000);

   B.Nab_za_kurs_d += new    Nab(S1.S_D);

    B.Nab_za_kurs_d += S2.S_D;

     B.K_D(2500);

      S1.vivod(); S2.vivod(); S3.vivod();
```

Существует еще одна форма описания события:

[спецификаторы] event <тип> <имя события>

{

add ***{Код добавления события в цепочку событий}***

remove ***{Код удаления события из цепочки событий}***

}

Средство доступа **add** вызывается в случае, когда с помощью операции "+=" в цепочку событий добавляется новый обработчик.

Средство доступа **remove** вызывается, когда с помощью операции "**-=**" из цепочки событий удаляется новый обработчик.

Средство доступа add или remove при вызове получает параметр, который называется **value**.

При использовании такой формы описания события можно задать собственную схему хранения и запуска обработчиков событий.

Например, для этого можно использовать массив, стек или очередь.

Следующий фрагмент кода показывает, как можно организовать вызов нужных методов в ответ на нажатие определенной клавиши.

```
public    delegate  void   s_n();

 class key
    {
       static   s_n[ ] sn =  new   s_n[3];

        public   static   event s_n   naz_klav
                {   add
                    { for (int i = 0; i < 3; i++)
                        { if (sn[i] == null)
                            { sn[i] = value; break; }
                        }
                    }
```

```
remove
        {
            for (int i = 0; i < 3; i++)
                {
                    if (sn[i] == value)
                        { sn[i] = null; break; }
                 }
         }
} // конец описания события
```

```
public   static   void   OnPressKey(ConsoleKey   k)
  {
        for (int i = 0; i < 3; i++)
        { if (("On" +k.ToString()) = =  sn[i].Method.Name)
                sn[i]( ); }
   }
    }

class    ff

  { public void OnDelete( ) { Console.WriteLine(« Delete в ff "); } }
```

```
class Program
    {
      static void OnDelete( )
      { Console.WriteLine("Delete!"); }

      static void OnInsert( )
            { Console.WriteLine("Insert!"); }

       static void OnHome( )
            { Console.WriteLine("Home!"); }
```

```
static   void   Main(string[ ] args)
      {  key.naz_klav  +=  OnDelete;
         key.naz_klav  +=  OnInsert;
         key.naz_klav  +=  OnHome;

         key.OnPressKey(Console.ReadKey(true).Key);

          key.naz_klav  - =  OnHome;

           ff    f =   new     ff();

            key.naz_klav += f.OnDelete;
```

Результат при нажатии Delete:

Delete!

```
    key.OnPressKey (Console.ReadKey(true).Key);
            Console.ReadKey();
        }
    }
```

Результат после нажатия Delete:

Delete!

Delete в ff

Правила обработки событий в среде .Net и использование встроенного делегата EventHandler.

Для создания .NET-совместимых событий рекомендуется следовать следующим правилам:

- имя событийного делегата рекомендуется заканчивать суффиксом **EventHandler**;
- имя обработчика события принято начинать с префикса **On**;
- обработчики событий должны иметь два параметра:
  - первый имеет тип **object** и задает ссылку на объект-источник события;

- второй параметр должен иметь тип **EventArgs** или производный от него, этот параметр задает остальную необходимую обработчику информацию.

Класс EventArgs используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля.

В класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

Если обработчикам события нужна дополнительная информация о событии (например, какая была нажата клавиша), необходимо создать класс, производный от EventArgs, и описать в нем все необходимые поля.

Например, в пример о студентах, наблюдающих за изменениями курса доллара, можно внести такие изменения:

Событийный делегат -

```
public delegate void   NabEventHandler(object   ob, EventArgs  arg);
```

Описание события –

```
public  event   NabEventHandler   Nab_za_kurs_d;
```

Метод, инициирующий событие –

```
public  void  K_D(double kd)
        { kurs_dol = kd;
           if (Nab_za_kurs_d != null)
                                Nab_za_kurs_d(this, EventArgs.Empty);
         }
```

Метод-обработчик события из класса Student –

```
public void OnS_D(object   ob,EventArgs  arg)
        {
            stip_dol = stip/((Bank)ob).KD;
        }
```

Тогда в методе Main будут использоваться следующие операторы:

```
B.Nab_za_kurs_d  += new     NabEventHandler(S1.OnS_D);
B.Nab_za_kurs_d  += S2.OnS_D;
```

Для иллюстрации передачи в методы-обработчики дополнительной информации переработаем пример с обработкой события нажатия клавиши.

Предусмотрим передачу вида нажатой клавиши.

```
class  KeyEventArgs :  EventArgs
    { public  ConsoleKey  klav;}
public  delegate void
                s_nEventHandler(object  ob, KeyEventArgs  args);
class  key
  {
    static  s_nEventHandler[ ] sn = new   s_nEventHandler[3];
```

```
public   static   event     s_nEventHandler    naz_klav
        {
           add { for (int i = 0; i < 3; i++)
                   { if (sn[i] == null)
                       { sn[i] = value; break; }
                    }
               }
```

```
remove
        {
            for (int i = 0; i < 3; i++)
            {
                if (sn[i] == value)
                { sn[i] = null; break; }
            }
        }
    }
```

```
public   void   OnPressKey(ConsoleKey k)
  {   KeyEventArgs   kk=new    KeyEventArgs();      kk.klav = k;

  for  (int i = 0; i < 3; i++)
          { if (("On" +k.ToString()) = =  sn[i].Method.Name)
                                              sn[i](this, kk); }
      }
   }

class ff { public void OnDelete(object  ob,KeyEventArgs   args)
    { Console.WriteLine(args.klav+"   в ff"); } }
```

```
static void OnDelete(object   ob, KeyEventArgs   args)
    { Console.WriteLine(args.klav + " удаление"); }
 static void OnInsert(object   ob, KeyEventArgs   args)
    { Console.WriteLine(args.klav + " вставка"); }
static void OnHome(object   ob, KeyEventArgs   args)
    { Console.WriteLine(args.klav + " домой"); }
```

Если обработчикам события не требуется дополнительная информация, для объявления события можно использовать стандартный тип делегата **System.EventHandler**.

Например, в примере о студентах-наблюдателях при использовании этого типа делегата становится ненужным объявление событийного делегата, а описание события принимает вид:

```
public event EventHandler Nab_za_kurs_d;
```

Пример (к лаб. работе 10).

Подготовить текстовый файл, содержащий информацию о сотрудниках фирмы: фамилия, название отдела, стаж, зарплата в $.

Разработать программу, которая

- считывает информацию из файла в массив объектов класса **Сотрудник.**

- предоставляет возможность пользователю выполнять следующие действия по нажатию определенной клавиши:

PageUp – зарплата у всех сотрудников увеличивается на 20%

PageDown – зарплата у всех сотрудников уменьшается на 10%

Delete – увольняются все сотрудники, стаж которых меньше 3 лет

Esc – выход из программы.

Каждое действие должно сопровождаться выводом результатов.

```
public   delegate  void   s_n();

  class key
      {
         static int n;
         static   s_n[ ]   sn;

          public    static   int   N
                  { set  { n = value;   sn = new   s_n[n]; } }

          public   static   event    s_n    naz_klav
                  {
```

```
add { for (int i = 0; i < n; i++)
                { if (sn[i] == null)
                    { sn[i] = value; break; }
                }
            }
remove
        {
            for (int i = 0; i < n; i++)
            {
                if (sn[i] == value)
                { sn[i] = null; break; }
            }       }       }
```

```
public static void OnPressKey(ConsoleKey k)
    {
        for (int i = 0; i < n; i++)
        { if (sn[i]!=null &&
            ("On" +k.ToString()) ==  sn[i].Method.Name) sn[i](); }
    }
}
```

```
class Sotrudnic
    {
        string fam, naz_otdel;
        int staz;
        double zarplata;
        public Sotrudnic(string fm, string nazotd, int stz, double zpl)
        {
            fam = fm; naz_otdel = nazotd; staz = stz;
            zarplata = zpl;
        }
```

```
public void vivod( )
        { Console.WriteLine(fam + " " + naz_otdel + "   " +
                                                    staz + " " + zarplata); }


  public void OnPageUp( )
          { zarplata = zarplata * 1.2; }


   public void   OnPageDown()
           { zarplata = zarplata * 0.85; }


    public int Staz { get { return staz; } }
    }
```

```
class Program
  {
     static bool p;
     static void    OnDelete()
     { Console.WriteLine("Увольнение молодых и
                                                  неопытных!");
}

 static    void   OnEscape()
       { p=false; }
```

```
static void Main(string[ ] args)
    {
        StreamReader f2 = new    StreamReader("baza2.txt");
        string s = f2.ReadLine();   int j = 0;
        while (s != null)
        {
            s = f2.ReadLine();
            j++;
        }
        f2.Close();
```

```
Sotrudnic[ ]   st = new Sotrudnic[j];

key.N = 2 * j+2;

 f2 = new StreamReader("baza2.txt");
            s = f2.ReadLine();   j = 0;

 while (s != null)
            {
              string[ ]    ss = s.Split(';');
st[j] = new   Sotrudnic(ss[0], ss[1], Convert.ToInt32(ss[2]),
                                                Convert.ToDouble(ss[3]));
```

```
key.naz_klav += st[j].OnPageDown;
key.naz_klav += st[j].OnPageUp;
        s = f2.ReadLine();
        j++;
    }
    f2.Close();

    foreach (Sotrudnic  sss  in  st) sss.vivod();

   key.naz_klav  +=  OnDelete; key.naz_klav  +=  OnEscape;
```

```
p = true;
        while (p)
        {
            key.OnPressKey(Console.ReadKey(true).Key);
            foreach (Sotrudnic sss in st) sss.vivod();
        }}}}
```