События (events)

- Конструкторы
- Деструкторы
- Константы
- Поля
- Методы
- Свойства
- Индексаторы
- Операторы
- События
- Делегаты
- Классы
- Интерфейсы
- Структуры

События позволяют классу или объекту уведомлять другие классы или объекты о возникновении каких-либо ситуаций.

Событие представляет собой сообщение, посылаемое объектом, чтобы сигнализировать о совершении какого-либо действия. Это действие может быть вызвано в результате взаимодействия с пользователем, например при нажатии кнопки мыши, или может быть обусловлено логикой работы программы.

Объект, вызывающий событие, называется **отправителем** события. Объект, который захватывает событие и реагирует на него, называется **получателем** события.

Часто используются другие термины - **издатель и подписчик.**

При обмене событиями классу отправителя событий не известен объект или метод, который будет получать (обрабатывать) сформированные отправителем события.

Необходимо, чтобы между источником и получателем события имелся посредник (или механизм подобный указателю).

В C# **события** неотъемлемо связаны с делегатами

События имеют следующие свойства:

- **Издатель** определяет момент вызова события, **подписчики** определяют предпринятое ответное действие.
- У события может быть несколько подписчиков. Подписчик может обрабатывать несколько событий от нескольких издателей.
- События, не имеющие подписчиков, никогда не возникают.
- В библиотеке классов **.NET Framework** в основе событий лежит делегат **EventHandler** и базовый класс **EventArgs**.

Элемент события определяется в классе с помощью ключевого слова **event**.

Когда компилятор обнаруживает ключевое слово **event** в классе, он создает элемент, например:

```
private EventNameHandler eh = null;
```

Компилятор создает при объявлении события два открытых метода **add_EventName** и **remove_EventName**.

Эти методы являются обработчиками событий, которые позволяют добавлять или удалять делегаты из делегата события **eh**. **Эти подробности скрыты от программиста**.

События это особый тип многоадресных делегатов, которые можно вызвать только из класса или структуры, в которой они объявлены (класс издателя).

Если на событие подписаны другие классы или структуры, их методы обработчиков событий будут вызваны когда класс издателя инициирует событие.

В следующем примере показан порядок объявления делегата событий:

```
public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
```

Зарезервированное слово **delegate** сообщает компилятору, что **AlarmEventHandler** является типом делегата.

По соглашению делегаты событий в .NET Framework имеют два параметра: источник, вызвавший событие, и данные для события.

Экземпляр делегата **AlarmEventHandler** можно привязать к любому методу, который соответствует его сигнатуре, такому как метод **AlarmRang** класса **WakeMeUp**, как показано в следующем примере.

```
public class WakeMeUp
{
   // AlarmRang has the same signature as AlarmEventHandler.
    public void AlarmRang(object sender, AlarmEventArgs e) {...};
    ...
}
```

Чаще всего события (events) используются в windows приложениях в которых допустим кнопка Button реагируя на события, выдают информацию на той же панели где они расположены (например щелчок мышкой).

Но события так же можно использовать и в консольных приложениях.

```
namespace ConsoleApplication3
{
    class Program
    {

        static void Main(string [ ] args)
        {
            car c = new car("BMW");
            car.MaxSpeed += new car.CarDelegate(car.Alert);
            for (int i = 0; i < 30; i++)
                c.SpeedUp();
        }
    }
```

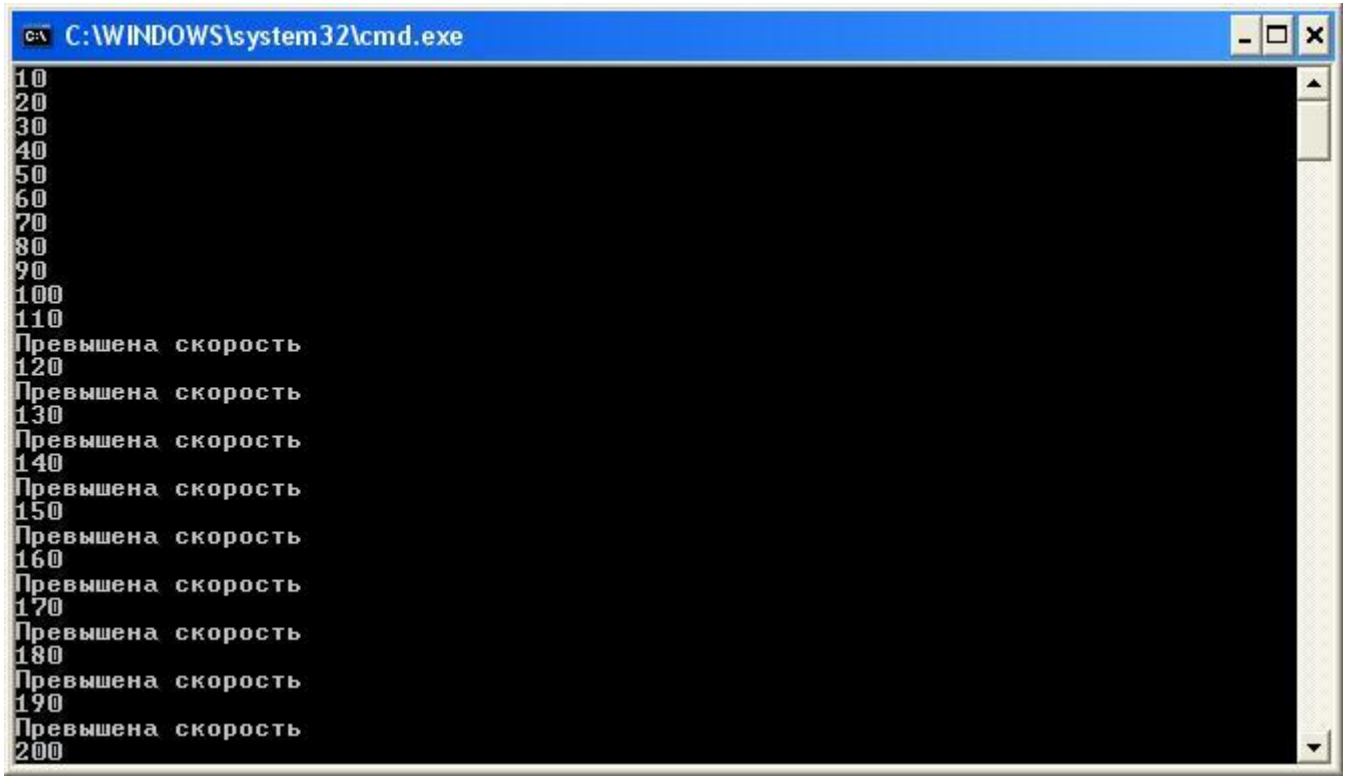
```
class car
{
    int speed = 0;
    public delegate void CarDelegate(string msg);
    public static event CarDelegate MaxSpeed;
    string    _name;

    public car(string name)
    {
        this._name = name;
    }
    public void SpeedUp()
    {
        this.speed += 10;
        Console.WriteLine(speed.ToString());
        if (speed > 100)
            MaxSpeed("Превышена скорость");
    }
    public static void Alert(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

В классе Car создаем делегат через который будет вызываться событие

Далее делаем вызов события когда скорость больше 100

Через делегат вызывается метод **Alert** который и выводит сообщение на консоль



Еще один простой пример:

Предположим, что стоит такая задача: определено три класса.

Первый класс будет считать до 100, используя цикл.

Два других класса будут ждать, когда в первом классе счетчик досчитает, например, до 71, и после этого каждый выведет в консоль фразу «Пора действовать, ведь уже 71!».

Проще говоря, при обнаружении значения 71, вызовутся по методу, соответственно для каждого класса.

```
class ClassCounter     //   Это класс - в котором производится счет.
{
     public delegate void MethodContainer();
     public event   MethodContainer onCount;  // Событие OnCount с типом делегата
                                               //  MethodContainer.

     public void Count()
      {
           for (int i = 0; i < 100; i++)
              {
                 if (i == 71)
                      {     onCount();  }
              }

      }

}
```

```
class Program
    {
        static void Main(string[] args)
            {
                ClassCounter Counter = new ClassCounter();

                Handler_I  Handler1 = new Handler_I( );
                Handler_II Handler2 = new Handler_II( );

               //Подписались на событие
                Counter.onCount += Handler1.Message;
                Counter.onCount += Handler2.Message;

              //Запустили счетчик
                Counter.Count();
            }
    }
```

```
class Handler_I //Это класс, реагирующий на событие (счет равен 71) записью строки в консоли.
{
    public void Message()
     { //Не забудьте using System
       //для вывода в консольном приложении
         Console.WriteLine("Пора действовать, ведь уже 71!");
      }
}
class Handler_II
  {
      public void Message()
     {
         Console.WriteLine("Точно, уже 71!");
    }
  }
```

События можно пометить как открытые **public**, закрытые **private**, защищенные **protected**, внутренние **internal** или **protectedinternal**.

Модификаторы доступа определяют порядок доступа к классу для пользователей класса.

ключевое слово	Описание
static	Делает событие доступным для вызова в любое время, даже если экземпляр класса отсутствует.
virtual	Позволяет производным классам переопределять поведение события при помощи ключевого слова ***override***
sealed	Указывает, что для производных классов событие более не является виртуальным.
abstract	Компилятор не создаст блоки методов доступа к событиям **add** и **remove** и, следовательно, производные классы должны предоставлять собственную реализацию.

Функциональные возможности события обеспечивают три взаимосвязанные элемента:

- класс, предоставляющий данные события,
- делегат события
- и класс, вызывающий событие.

В среде .NET Framework используется соглашение по присвоению имен классам и методам, относящимся к событиям.

Чтобы класс мог вызывать событие с именем EventName, необходимы следующие элементы:

- Класс, содержащий данные события, именуемый как **EventNameEventArgs**. Этот класс должен наследоваться от **System.EventArgs**.
- Делегат для события, именуемый как **EventNameEventHandler**.
- Класс, вызывающий событие. Этот класс должен предоставить объявление события (**EventName**) и метод, инициирующий событие (**OnEventName**).

Класс данных события и класс делегата события могут быть уже определены в библиотеке классов .NET Framework или в библиотеке классов независимых разработчиков. В этом случае не требуется определять эти классы.

Например, если событие не использует пользовательские данные, то можно использовать System.EventArgs для данных события и System.EventHandler для делегата.

Подписка и отмена подписки на события

Необходимость подписки на событие, опубликованное другим классом, может возникнуть когда требуется написать пользовательский код, вызываемый при инициировании такого события.

Например, можно подписаться на событие кнопки **click**, чтобы приложение выполняло некоторое действие при нажатии пользователем на кнопку.

Подписка на события в среде IDE Visual Studio

1. Если окно **Свойства** закрыто, в представлении **Конструктор** щелкните правой кнопкой мыши форму или элемент управления, для которого требуется создать обработчик событий, и выберите пункт **Свойства**.
2. Вверху окна **Свойства** щелкните значок **События**.
3. Дважды щелкните событие, которое требуется создать, например событие **Load**.

Visual C# создаст пустой метод обработчика событий и добавит его в код.

Код можно также добавить вручную в представлении **Код**.

Например, следующие строки кода объявляют метод обработчика событий, который будет выполнен при инициировании классом **Form** события **Load**.

```
private void Form1_Load(object sender, System.EventArgs e)
{
  // Add your form load event handling code here.
}
```

Строка кода, требуемая для подписки на событие, также создается автоматически в методе **InitializeComponent** в файле **Form1.Designer.cs** проекта. Она имеет следующий вид.

```
this.Load += new System.EventHandler(this.Form1_Load);
```

```
public class SampleEventArgs
{
   public SampleEventArgs(string s) { Text = s; }
   public String Text {get; private set;}          // readonly
}

public class Publisher
{
   public delegate void SampleEventHandler(object sender, SampleEventArgs e);

   public event SampleEventHandler SampleEvent;  // Объявляем событие.

 // Wrap the event in a protected virtual method
 // to enable derived classes to raise the event.
   protected virtual void RaiseSampleEvent()
      {
        if (SampleEvent != null)
                SampleEvent(this, new SampleEventArgs("Hello"));
      }
 }
```

Подписка на события программными средствами

1. Определите метод обработчика событий, подпись которого соответствует подписи делегата для события. Например, если событие основано на типе делегата **EventHandler**, то следующий код представляет заглушку метода:

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
  // Do something useful here.
}
```

2. Чтобы присоединить обработчик событий к событию, используйте оператор назначения сложения (**+=**).

В следующем примере предположим, что объект с именем **publisher** имеет событие с именем **RaiseCustomEvent**. Обратите внимание, что для класса подписчика требуется ссылка на класс издателя, чтобы подписаться на его события.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Приведенный выше синтаксис появился только в C# 2.0. Он в точности соответствует синтаксису C# 1.0, в котором при помощи ключевого слова **new** должен быть явно создан инкапсулирующий делегат:

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

Для добавления обработчика событий можно также использовать лямбда-выражение.

```
public Form1()
{
  InitializeComponent();
  this.Click += (s,e) => {
          MessageBox.Show( ((MouseEventArgs)e).Location.ToString());
                        };
}
```

Подписка на события при помощи анонимного метода

Если не нужно будет позже отменять подписку на событие, можно использовать оператор назначения сложения (+=) для прикрепления к событию анонимного метода.

В следующем примере предположим, что объект с именем **publisher** имеет событие с именем **RaiseCustomEvent** , и что класс **CustomEventArgs** также был определен и содержит некие относящиеся к событию сведения. Обратите внимание, что для класса подписчика требуется ссылка на **publisher**, чтобы подписаться на его события.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Отменить подписку на событие не так просто, если для подписки на него использовалась анонимная функция. Чтобы отменить подписку в этом случае, необходимо вернуться к коду, в котором была выполнена подписка на событие, сохранить анонимный метод в переменной делегата, а затем добавить делегат к событию.

Как правило, не рекомендуется использовать анонимные функции для подписки на события, если предполагается, что в будущем будет нужно отменять подписку на событие.

Отмена подписки

Чтобы предотвратить вызов обработчика событий при инициировании события, подписку на событие необходимо отменить. Во избежание утечки ресурсов отменять подписку на события следует до удаления объекта подписчика.

До тех пор, пока подписка на событие не отменена, делегат многоадресной рассылки, лежащий в основе события в публикующем объекте, будет ссылаться на делегат, инкапсулирующий обработчик событий подписчика.

Если ссылка присутствует в публикующем объекте, объект подписчика не будет удален при сборке мусора

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

Если подписка на событие отменена для всех подписчиков, экземпляр события в классе издателя получает значение **null**.

Инициирование события

После определения реализации события, необходимо определить, когда следует **инициировать** событие.

Событие инициируется вызовом защищенного метода **OnEventName** в классе, определяющем событие, или в производном классе.

Метод **OnEventName** вызывает событие посредством вызова делегатов, передавая все характерные для события данные. Методы делегата события могут выполнять действия для события или обрабатывать характерные для события данные.

Пользовательские делегаты событий необходимы только в случаях, когда событие создает данные для события. Многие события, включая некоторые события пользовательского интерфейса, например щелчки мышью, не создают данных для события.

В таких ситуациях делегат события, предоставляемый библиотекой классов для события без данных, **System.EventHandler** является целесообразным. Объявление делегата следующее:

```
delegate void EventHandler(object sender, EventArgs e);
```

.NET Framework 2.0 представляет общую версию данного делегата **EventHandler<TEventArgs>**.

В следующей процедуре показано добавление событий, соответствующих стандартному шаблону .NET Framework для пользовательских классов и структур.

Порядок публикации событий, основанных на шаблоне EventHandler

1. (Пропустите этот шаг и перейдите к шагу 3a, если не требуется передавать с событием пользовательские данные.)

   Объявите класс для пользовательских данных в области, видимой для классов издателя и подписчика. Затем добавьте необходимые члены для хранения данных пользовательских событий. В данном примере возвращается простая строка.

```
public class CustomEventArgs : EventArgs
{
  public CustomEventArgs(string s) { msg = s; }
  private string msg;
  public string Message { get { return msg; } }
}
```

2. (Пропустите данный шаг, если используется общая версия **EventHandler<TEventArgs>** .)

Объявите делегат в своем классе публикации. Назначьте ему имя, заканчивающееся на **EventHandler**. Второй параметр задает ваш тип **EventArgs**.

```
public delegate void CustomEventHandler(object sender, CustomEventArgs a);
```

3. Объявите событие в своем классе публикации с помощью одного из следующих действий.

- Если пользовательский класс **EventArgs** отсутствует, ваш тип **Event** представляет собой не являющийся общим делегат **EventHandler**. Этот делегат не нужно объявлять, так как он уже объявлен в пространстве имен **System**, добавленном при создании проекта C#. Добавьте следующий код в класс издателя.

```
public event EventHandler RaiseCustomEvent;
```

- При использовании неуниверсальной версии типа **EventHandler** и наличии пользовательского класса, производного от типа **EventArgs**, объявите событие внутри класса публикации и используйте делегат из пункта 2 в качестве типа.

```
public event CustomEventHandler RaiseCustomEvent;
```

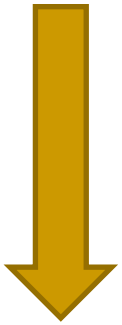
- Если используется универсальная версия, то пользовательский делегат не требуется. Вместо этого в классе публикации необходимо задать тип события как **EventHandler<CustomEventArgs>**, заключив имя пользовательского класса в угловые скобки.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

```
namespace DotNetEvents
{
   using System;
   using System.Collections.Generic;

   // Класс для хранения пользовательской информации

   public class CustomEventArgs : EventArgs
   {
    public CustomEventArgs(string s) { message = s; }
    private string message;
    public string Message { get { return message; } set { message = value; } }
   }
```



```
// Класс для публикации событий

class Publisher
  {
    public event EventHandler<CustomEventArgs> RaiseCustomEvent; // Декларирует событие

    public void DoSomething()
     {   // Здесь код, который что-то делает полезное, затем вызывает событие.
         // Можно создать событие, прежде чем выполнять блок кода
        OnRaiseCustomEvent(new CustomEventArgs("Did something"));
     }

     protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
     {
       // Make a temporary copy of the event to avoid possibility of a race condition if the last subscriber unsubscribes
       // immediately after the null check and before the event is raised.
       EventHandler<CustomEventArgs> handler = RaiseCustomEvent;
       if (handler != null)  // Событие null если нет подписчиков
        {
          e.Message += String.Format(" at {0}", DateTime.Now.ToString());
          handler(this, e);
        }
     }
  }
```

```
//Класс, который подписывается на события

class Subscriber
  {
      private string id;

      public Subscriber(string ID, Publisher pub)
        { id = ID;              // Подписаться на событие (C# 2.0 syntax)
          pub.RaiseCustomEvent += HandleCustomEvent;
        }

       // Определить какие действия необходимо предпринять при
       // возникновении события

      void HandleCustomEvent(object sender, CustomEventArgs e)
      {
        Console.WriteLine(id + " received this message: {0}", e.Message);
      }

}
```

```
class Program
  {
      static void Main(string[] args)
        {
          Publisher pub = new Publisher();                // Публикация
          Subscriber sub1 = new Subscriber("sub1", pub); // Подписка
          Subscriber sub2 = new Subscriber("sub2", pub);
          pub.DoSomething(); // Вызов метода, который вызывает событие

          Console.WriteLine("Press Enter to close this window.");
          Console.ReadLine();
        }
   }
}
```

События и делегаты. Различия

Так в чем же разница между событиями и делегатами в .NET?

При объявлении события очевидно его строгое соответствие определенному делегату:

```
public static event    System.EventHandler    xEvent;
```

System.EventHandler – это *ТИП ДЕЛЕГАТА*!

Оператор, который обеспечивает процедуру "подписания на уведомление", полностью соответствует оператору модификации многоадресного делегата. Аналогичным образом дело обстоит и с процедурой "отказа от уведомления":

```
BaseClass.xEvent += new System.EventHandler(this.MyFun);
BaseClass.xEvent –= new System.EventHandler(xxx.MyFun);
```

За операторными функциями += и –= скрываются методы классов-делегатов (в том числе и класса-делегата System.EventHandler).

Более того. Если в последнем примере в объявлении события ВЫКИНУТЬ ключевое слово event

```
public static event System.EventHandler xEvent;
```

и заменить его на

```
public static System.EventHandler xEvent;
```

System.EventHandler (это класс-делегат!), то все будет происходить, как и раньше! Вместо пары СОБЫТИЕ-ДЕЛЕГАТ будет работать пара ДЕЛЕГАТ-ДЕЛЕГАТ

Таким образом, функционально **событие** является всего лишь разновидностью **класса-делегата**, главной задачей которого является обеспечение строгой привязки делегата к соответствующему событию

Модификатор event вносит лишь незначительные синтаксические нюансы в использование этого МОДИФИЦИРОВАННОГО делегата — чтобы хоть как-нибудь различать отправителя и получателя события.