Темы урока

[**События**](#_6zj5vaog9t7n) **1**

[Демонстрация в коде](#_2lqda695ib61) 2

[Освежим знания о делегатах](#_a3o3bg8is0x9) 2

[Переходим к событиям](#_oajjy7h3g17h) 3

[Вызов событий](#_arvqnd3wgy66) 3

[Подписка на события объекта](#_6e7b7eig5tji) 4

[Самостоятельная работа](#_5tdoa0uvrt1v) 5

[EventArgs, EventHandler<T>](#_wrcbr6udqem7) 5

[Самостоятельная работа](#_t0obinzh7n6y) 6

[**Chatbot: Интерфейсы и классы**](#_36j67pvhcg82) **6**

[**Домашнее задание**](#_a0br4obaio46) **7**

[Chatbot](#_53eiygg2rfd6) 7

[События (опционально)](#_igevwzpayrjt) 7

# События

Событие — автоматическое уведомление о том, что произошло некоторое действие.

События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события. Обработчики событий обычно представлены делегатами..

События являются членами класса и объявляются с помощью ключевого слова event.

Чаще всего для этой цели используется следующая форма:

|  |
| --- |
| модификатор\_доступа event делегат\_события имя\_события; |

например:

|  |
| --- |
| public event WorkPerformedEventHandler WorkPerformed; |

где, WorkPerformed - это событие, а WorkPerformedEventHandler - это делегат.

## 

## Демонстрация в коде

#### Освежим знания о делегатах

Давайте напишем простой тип делегата для выполнения какой-то длительной работы.

* Открыть проект **L17\_C01\_events\_demo**
* Определяем перечисление  
   public WorkType { Work, DoNothing }
* Определяем в классе Program тип делегата  
   public delegate void WorkPerformedEventHandler(int hours, WorkType workType);
* Определим три условно разные метода, подходящие типу делегата по сигнатуре - это будут конкретные исполнители работы заданного типа и продолжительности.  
    
  Создаём в классе Program 3 статических метода WorkPerformed1, ...2 и ...3:  
   static void WorkPerformed1(int hours, WorkType workType) { вывод в консоль }
* В методе Main
  + Cоздаем три делегата de1, del2 и del3 с каждым отдельным методом WorkPerformed1, ...2 и ...3:
    - var del1 = new WorkPerformedEventHandler(WorkPerformed1);  
      и, соответственно, del2… WorkPerformed2, del3… WorkPerformed3.
  + Устанавливаем del1 += del2 + del3;
  + Вызываем del1(1, WorkType.Work);
* Демонстрируем вывод в консоль трёх сообщений, каждый от своей имплементации.
* **Но они ничего не возвращают! Давайте поменяем тип возвращаемого значения в типе делегата с void на int, чтобы дать возможность исполнителю отчитаться как-то о проделанной работе.**
  + public delegate **int** WorkPerformedEventHandler
* Соответственно меняем методы статические методы WorkPerformed1, ...2 и ...3.  
  Будем возвращать hours + 1, 2 и 3, соответственно.
* Сохраняем результат вызова на исполнение делегата del1 в переменную и выводим её на экран:  
   int finalResult = del1(1, WorkType.Work);  
   Console.WriteLine(finalResult);
* Запускаем, демонстрируем “4” в качестве результата.  
  Напоминаем, что делегаты с более чем одной связанное функцией возвращают результат вызова последней.

Финалом будет версия кода **L17\_C02\_events\_demo\_no\_good**.

#### Переходим к событиям

Мы идём дальше, уже приводя код к **L17\_C03\_events\_demo\_worker**.

* Создадим класс Worker, который должен делать работу
* Добавляем метод public **void** DoWork(int hours, WorkType workType);
* Положим определение типа делегата прямо рядом с классом, событию которого ссылается на этот делегат
  + Переносим  
    public delegate void WorkPerformedHandler(int hours, WorkType workType);.  
    в файл класса Worker (над классом)
  + Можно его перенести в отдельный класс, так как на уровне компиляции он превратится в отдельный класс, однако, мне нравится располагать его так.
* Теперь определим в классе наше событие:  
   public **event** WorkPerformedHandler WorkPerformed;
* Давайте также предположим, что когда работа полностью сделана, мы также хотим вызвать отдельное событие, сигнализирующее об этом. Для этого я добавлю ещё одно событие, которое назову WorkCompleted:  
   public event EventHandler WorkCompleted;,  
  В этом событии я не собираюсь передавать каких-то значений, я просто хочу отметить факт, то работа завершена.
  + Можно перейти к определению EventHandler и увидеть, что это делегат.

#### Вызов событий

**Обращаемся к слайдам**

* Вызов события - это основополагающая операция в концепции событий. Если событие не вызвать, то все, кто его ждут, все подписавшиеся на него слушатели, так и не узнают, что оно произошло.
* Вызов события происходит тем же способом, что и вызов делегата, вы вызываете его как метод.
* Однако, прежде чем вызвать событие, необходимо посмотреть, есть хоть что-то в списке вызовов у нашего события, иначе получится, что вызывать нам совершенно нечего. Помните, что делегат это ссылка на метод. И запуская на выполнение делегат, мы на самом деле вызываем метод, однако если список вызовов пуст, произойдёт исключение.
* Вызов события лучше осуществлять не напрямую, а через метод с префиксом “On”.
  + При этом можно сделать его **protected virtual**, чтобы дать возможность его переопределять в производных классах, если это будет необходимо.

Потом возвращаемся к коду:

* Давайте добавим вызов событий внутри нашего метода DoWork
* Идея будет такая - каждый час мы будем вызывать событие WorkPerformed, сообщающее о том, что такая-то работа проделана, что-то наподобие микроменеджмента.
  + Добавим цикл по часам и внутри будем вызывать метод OnWorkPerformed.
  + Добавим метод OnWorkPerformed, в котором мы будем проверять событие WorkPerformed, и если оно не равно null, вызывать его с параметрами переданными в метод OnWorkPerformed.
* Когда мы закончим и выйдем за пределы цикла, мы вызовем событие WorkCompleted. Также через protected virtual void OnWorkCompleted уже без параметров. А внутри него после проверки на null вызываем  
   WorkCompleted(this, EventArgs.Empty);

#### Подписка на события объекта

* Возвращаемся в метод Main класса Program
* Создаём объект класса Worker
* Привязываем к событиям обработчики
* Вызываем DoWork
* Запускаем.
* Наслаждаемся!

Код доступен **L17\_C03\_events\_demo\_worker**.

## 

## Самостоятельная работа

Убедиться, что случайные данные плохо упаковываются архиваторами (на примере Zip).

Для этого мы напишем генератор случайных данных, который выдавать **запрошенное** число произвольных байтов в виде массива.

Затем сохраним эти байты в бинарном виде в файл, заархивируем его и сравним размер архива с размером оригинального файла.

1. Генерация должна происходить в классе **RandomDataGenerator** в единственном публичном методе   
    public byte[] GetRandomData(int dataSize, int bytesDoneToRaiseEvent)  
   1. Первый параметр **dataSize** - размер массива в байтах
   2. Второй параметр **bytesDoneToRaiseEvent** - число байт, после очередной генерации которых надо вызвать событие **RandomDataGenerated**, связанное с делегатом типа:  
        
      public delegate void **RandomDataGeneratedHandler**(int bytesDone, int totalBytes);
   3. После завершения генерации необходимо вызвать событие **RandomDataGenerationDone**, связанное с делегатом типа **EventHandler**.
2. В основном потоке программы подписаться на оба события.
   1. В обработчике **RandomDataGenerated** необходимо выводить строку по примеру:  
      Generated XXX from YYY byte(s)...
   2. В обработчике **RandomDataGenerationDone**:  
      Generation DONE
3. Вывести получившийся массив на экран в виде Base64-строки, воспользовавшись Convert.ToBase64String()
4. Сохранить получившийся массив в файл **в бинарном виде**.
5. Средствами ОС создать заархивировать файл и сравнить размер.

Решение: **L17\_C04\_events\_SW**.

## EventArgs, EventHandler<T>

**По слайдам**

Стандартным способом передачи параметров в обработчик событий является объект класса EventArgs или его наследного класса.

.NET включает в себя обобщённый класс EventHandler<T>, который может использоваться вместо собственного делегата.

На примере привести **L17\_C03\_events\_demo\_worker** к **L17\_C05\_events\_eventargs\_final**.

## Самостоятельная работа

Замените собственный делегат RandomDataGeneratedHandler на встроенный EventArgs<T>.

Внесите необходимые изменения в программу, чтобы она снова компилировалась и работала верно.

Решение: **L17\_C06\_events\_eventargs\_SW**.

# Chatbot: Интерфейсы и классы

Живое обсуждение в классе возможной компоновки задач приложения.

Основные идеи:

* Reminder.Storage.Core
  + Библиотека с описанием интерфейсов и классов, которые будут использоваться конкретными реализациями хранилища данных. Здесь будут описаны
    - класс ReminderItem,
    - перечисление ReminderItemStatus,
    - интерфейс IReminderStorage.
  + Ей в пару будет создана сборка Reminder.Storage.Core.Tests для тестов логики классов (если она понадобится).
* Reminder.Storage.InMemory
  + Библиотека с реализацией хранилища данных в памяти. Т.е. здесь будет находиться класс InMemoryReminderStorage реализации интерфейса IReminderStorage.
  + Ей в пару будет создана сборка Reminder.Storage.InMemory.Tests для тестов логики хранилища.
* Reminder.Domain
  + Основная библиотека логики, которая будет отвечать за периодическую проверку, а не пора ли послать напоминание, и если пора, обеспечивать отправку.

# 

# Домашнее задание

## Chatbot

Написать интерфейсы и классы (со взаимосвязями) проекта chatbot, которые мы обсудили во время классной работы.

## События (опционально)

Написать класс FileWriterWithProgress, у которого был бы один метод  
public void WriteBytes(string fileName, byte[] data, float percentageToFireEvent)

* первый параметр - это имя файла,
* второй параметр - это сам массив для записи
* третий параметр - это процентная величина для вызова периодического события о прогрессе (0 < percentageToFireEvent < 1) - см. пример ниже

Класс должен предоставлять 2 типа события

* WritingPerformed достигнут прогресс записи, кратный кратный параметру percentageToFireEvent
* WritingCompleted достигнут конец записи.

Примеры:

var writer = new FileWriterWithProgress();

writer.WriteBytes(data, 0.1);

// будет 11 событий - 10 событий WritingPerformed при достижении 10%, 20%, …, 100% записи + 1 событие WritingCompleted при завершении.

writer.WriteBytes(data, 0.15);

// будет 7 событий - 6 событий WritingPerformed при достижении 15%, 30%, …, 90% записи + 1 событие WritingCompleted при завершении.