

В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

# Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

## Модуль 3. Лекция 2а

### События

# Оповещение Объектов о Событиях

При написании программ возникает сценарий, когда одни объекты должны получать оповещения от других.

При этом:

- Постоянный опрос объекта-издателя о новостях/обновлениях осуществлять неудобно;
- Отправлять информацию всем потенциальным объектам-подписчикам – неподходящее решение.
- Типы объектов-издателей и объектов-подписчиков должны быть независимы друг от друга.

Необходим механизм, позволяющий оповещать о происходящем только те объекты, который действительно нуждаются в получении информации.

*Подумайте, какие средства языка могут использоваться при решении этой задачи. Почему?*

# Терминология

Обозначим основные понятия, возникающие в контексте данной задачи:

**Издатель** (publisher) – тип, экземпляры которого генерируют события (raising an event). Позволяет другим объектам подписываться на рассылку событий и отписываться от неё.

**Подписчик** (subscriber) – тип, экземпляры которого способны подписываться на рассылку событий и обрабатывать связанные с ней данные.

**Обработчик события** (event handler) – метод обработки события. Как правило, используется объектом-подписчиком.

**Данные события** (event arguments) – информация, рассылаемая издателем всем подписчикам при возникновении события.

# Постановка Задачи

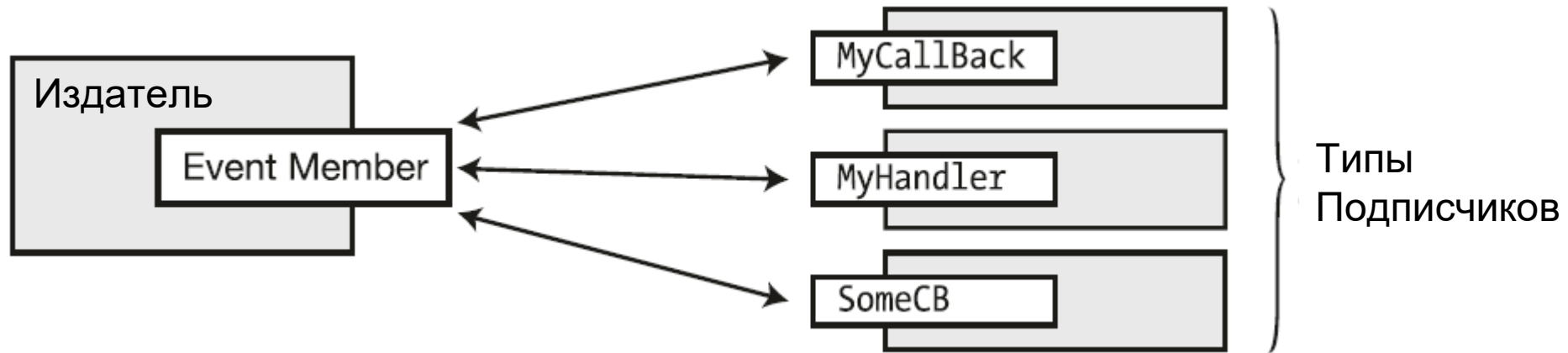
## Издатель:

- Позволяет подписываться и отписываться на рассылку событий;
- Принимает циклически консольные команды до ввода команды «exit» (завершение работы). Если принята команда «notify», все подписчики получают текущее время в качестве информации.

## Подписчик:

- Имеет собственное имя (строку);
- Подписывается на рассылку событий – имеет метод, соответствующий формату передаваемого сообщения;
- В момент получения сообщения от издателя обрабатывает его – выводит текст со своим именем и содержащиеся в сообщении данные.

# Схема Взаимодействия Издателя и Подписчиков

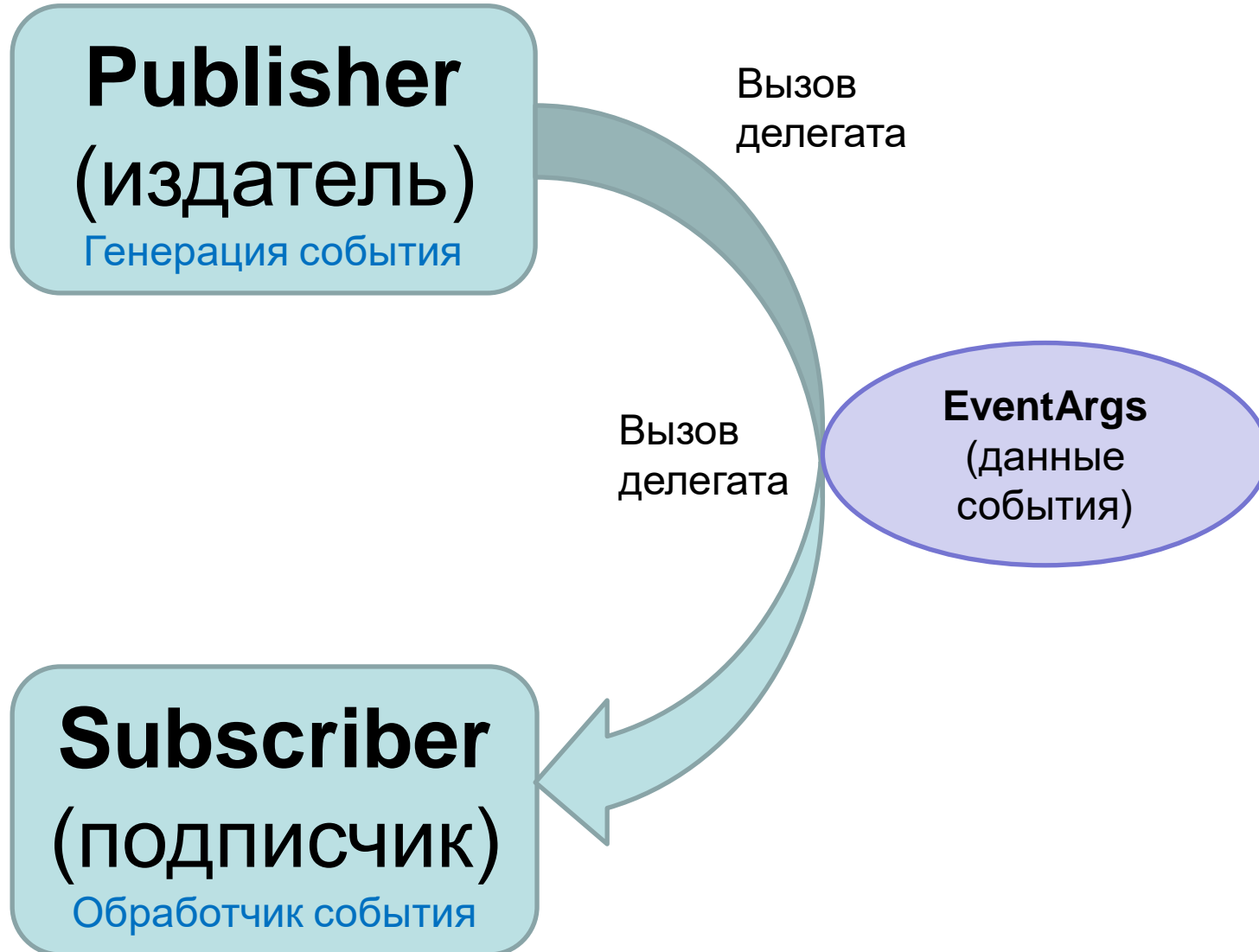


1. Издатель определяет событие (как член класса).

2. Подписчики определяют методы-обработчики, которые должны быть вызваны при возникновении события, и подписывают их на событие издателя.

3. Издатель инициирует возникновение события и, как следствие, вызываются все обработчики подписчиков события.

# Упрощённая Схема Задачи



# Решение 1: Использование Делегатов

Т.к. для передаваемое событие содержит данные определённого формата, а оповещение сводится к вызову реакции на событие со стороны подписчиков, подходящим решением задачи могут быть делегат-типы:

- Сигнатура делегат-типа задаёт контракт передаваемых данных между издателем и подписчиком;
- Возможность делегатов хранить ссылки на несколько методов позволяет сохранять всех подписчиков в одном списке вызовов.

*Подумайте, как можно было бы реализовать подобную схему в объектно-ориентированной парадигме при отсутствии поддержки делегатов (например, в Java или C++).*

# Решение 1. Часть 1 – Издатель

```
using System;
```

```
public class Publisher
```

```
{
```

```
    public Action<DateTime> notificationAppeared;
```

```
    public void HandleCommands()
```

```
    {
```

```
        string command;
```

```
        do
```

```
        {
```

```
            command = Console.ReadLine();
```

```
            if (command == "notify" && notificationAppeared != null)
```

```
            {
```

```
                notificationAppeared(DateTime.Now);
```

```
            }
```

```
        } while (command != "exit");
```

```
    }
```

```
}
```

1) Делегат, на который подписываются подписчики.

2) Циклическая обработка команд – при вводе notify всем подписчикам (если такие есть) будет разослано время возникновения события.



# Решение 1. Часть 2 – Подписчик

```
public class Subscriber  
{
```

```
    public string Name { get; init; }
```

```
    public Subscriber(string name) => Name = name;
```

```
    public void NotificationEventHandler(DateTime eventArgs)  
        => Console.WriteLine($"{Name}: received notification " +  
                               $"on {eventArgs.ToShortDateString()}");
```

```
}
```

1) Имя подписчика.

2) Метод соответствующий  
сигнатуре события издателя  
для обработки сообщений.

# Решение 1. Часть 3 – Основная Программа

```
class Program
{
    static void Main()
    {
        Publisher publisher = new Publisher();
        Subscriber subscriber = new Subscriber("Alex");
        publisher.notificationAppeared += subscriber.NotificationEventHandler;
        publisher.HandleCommands();
        // Код на строке ниже не скомпилируется для события:
        // publisher.notificationAppeared = null;
    }
}
```

Подписка объекта-подписчика  
на издателя осуществляется  
аналогичным образом.

# Проблемы Решения 1

Хотя данное решение позволяет выполнить задачу, оно имеет ряд недостатков:

- Вызов делегата издателя может быть осуществлён любым кодом без ограничений;
- Подписчики могут напрямую заменять список вызовов делегата через =;
- Любой код может полностью очистить список подписчиков, присвоив делегату null.

Вывод: требуется инкапсулировать (защитить) делегат так, чтобы можно было контролировать доступ к нему.

# События. Ключевое Слово **event**

Для упрощения реализации сценариев обработки событий было добавлено **ключевое слово event**, которое позволяет объявить инкапсулированное поле делегат-типа с использованием следующего синтаксиса:

*[Модификаторы] event <Делегат-Тип> <Идентификатор>;*

**Объявление поля делегат-типа как события** вводит следующие ограничения:

- События могут быть вызваны только внутри того типа, в котором они объявлены (у наследников доступа тоже нет!);
- Для подписчиков извне доступны только операции подписки/отписки (**+=** и **-=**).

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/>

# Решение 2: События. Часть 1 – Издатель

```
using System;
```

```
public class Publisher  
{
```

```
    public event Action<DateTime> notificationAppeared;
```

```
    public void HandleCommands()  
{
```

```
        string command;
```

```
        do
```

```
        {
```

```
            command = Console.ReadLine();
```

```
            if (command == "notify" && notificationAppeared != null)
```

```
            {
```

```
                notificationAppeared(DateTime.Now);
```

```
            }
```

```
        } while (command != "exit");
```

```
    }
```

```
}
```

Поле делегат типа теперь объявляется как событие, что добавляет необходимую инкапсуляцию.

Публикация события синтаксически не отличается от вызова делегата.

# Решение 2: События. Часть 2 – Подписчик

```
public class Subscriber
{
    public string Name { get; init; }

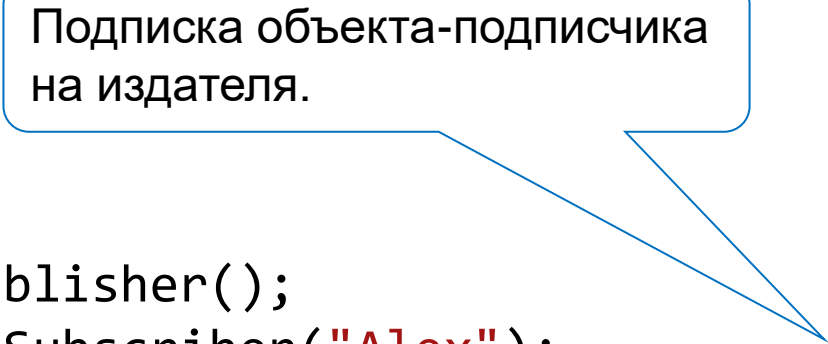
    public Subscriber(string name) => Name = name;

    public void NotificationEventHandler(DateTime eventArgs)
        => Console.WriteLine($"{Name}: received notification " +
                               $"on {eventArgs.ToShortDateString()}");
}
```

Метод-обработчик события синтаксически никак не меняется, т. к. событие основывается на том же делегат-типе.

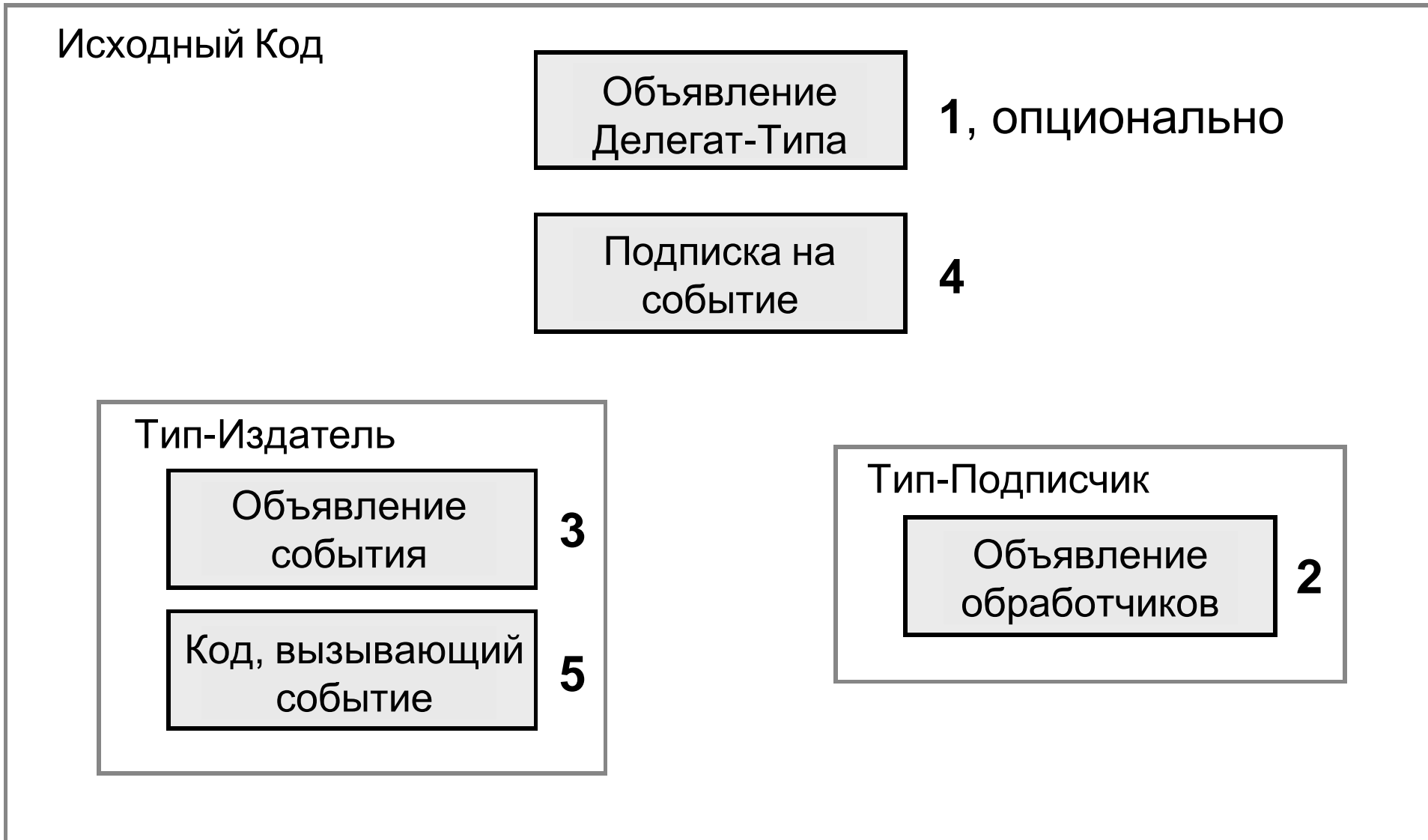
# Решение 2: События. Часть 3 – Основная Программа

```
class Program
{
    static void Main()
    {
        Publisher publisher = new Publisher();
        Subscriber subscriber = new Subscriber("Alex");
        publisher.notificationAppeared += subscriber.NotificationEventHandler;
        // Метод-обработчик подписчика вызывается на каждый ввод строки "notify".
        publisher.HandleCommands();
    }
}
```



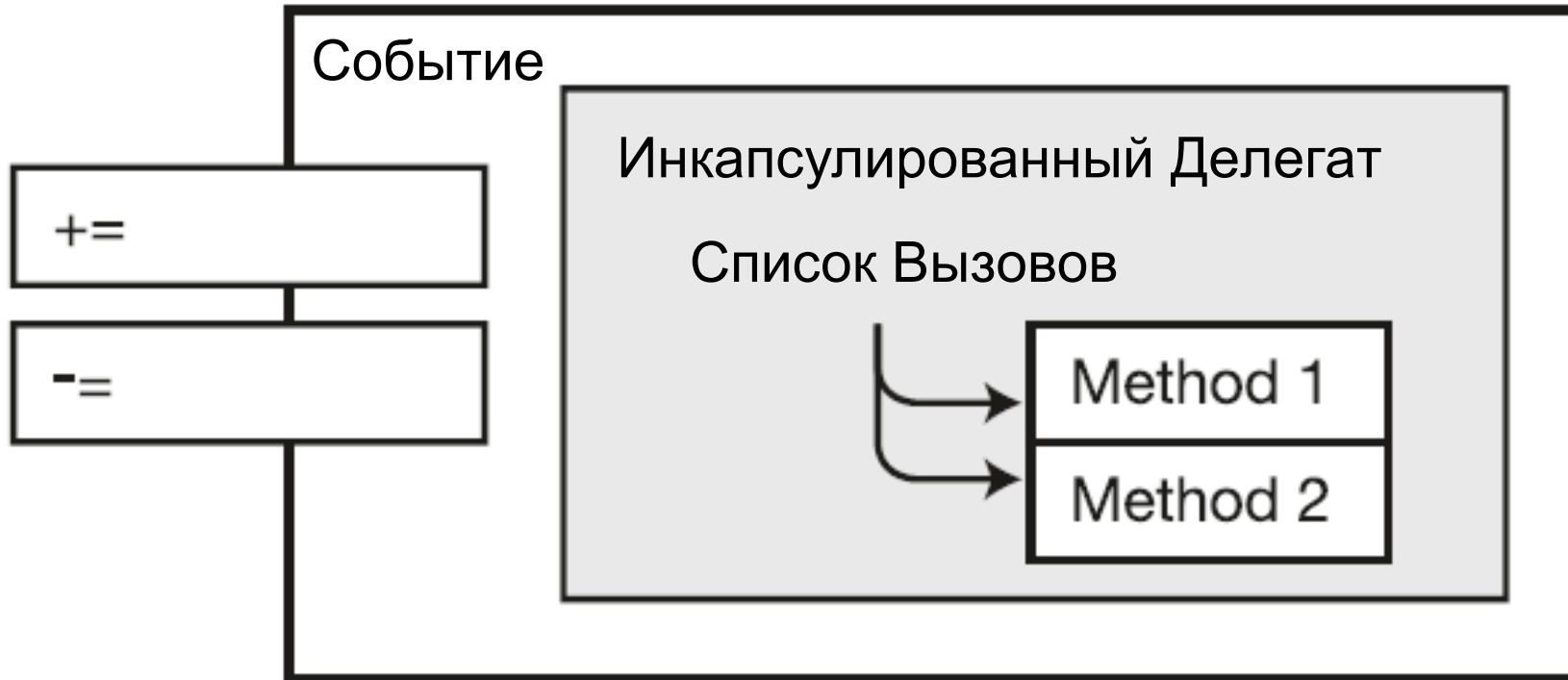
Подписка объекта-подписчика на издателя.

# Схема: Использование Событий в Коде

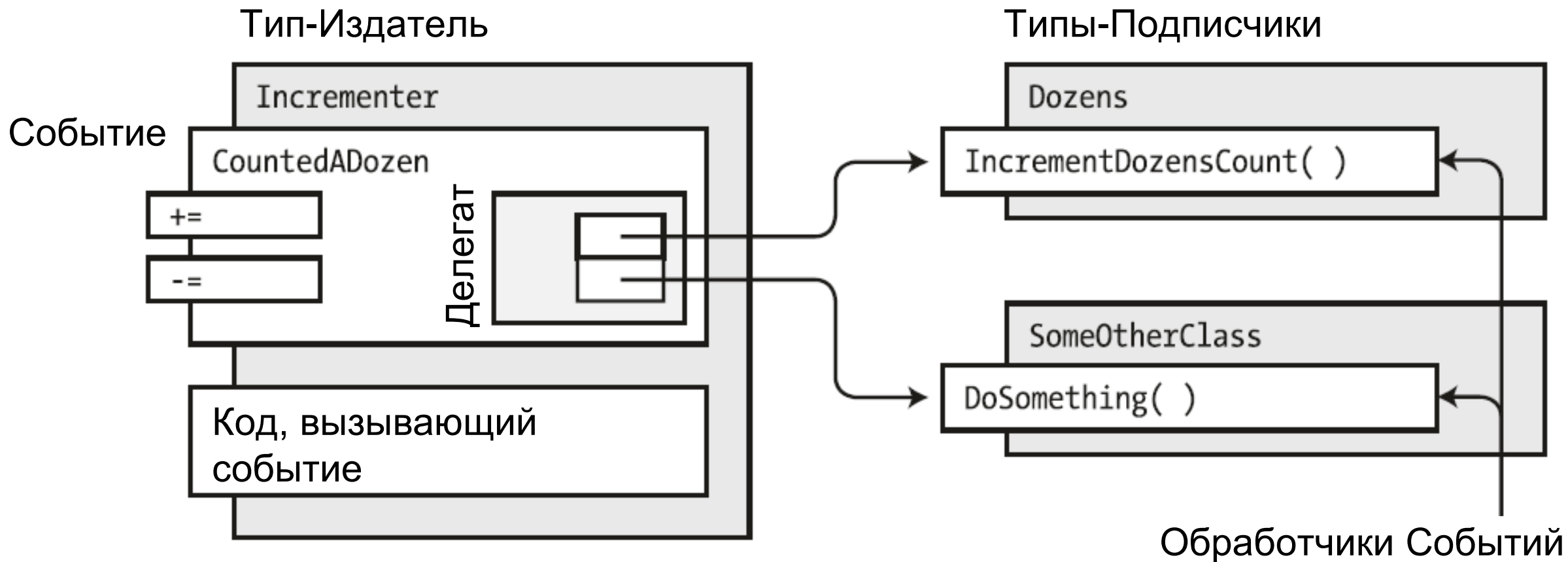




# Схема: Событие как Инкапсулированный Делегат



# Схема: Возникновения События, Обработка



# Возникновение Событий и Многопоточность

*Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.*

В прошлом примере вызов события осуществлялся с предварительной проверкой на null:

```
if (notificationAppeared != null)           // Проверить наличие подписчиков.  
    notificationAppeared(DateTime.Now);     // Вызвать событие.
```

Тем не менее, такой код небезопасен в многопоточной среде – возможен сценарий, когда после проверки на null список вызовов будет очищен из другого потока, а вызов делегата приведёт к `NullReferenceException`.

Более безопасным вариантом является вызов с использованием операции `?.`:

```
notificationAppeared?.Invoke(DateTime.Now);
```

**Что эквивалентно:**

```
object temp = notificationAppeared;  
if (temp != null)  
    temp.Invoke(DateTime.Now);
```

*Помните, что такая логика тоже не гарантирует 100% безопасность.*

# Стандартный Шаблон Генерации Событий .NET

В целях унификации работы с событиями Microsoft рекомендует использовать библиотечный делегат-тип в качестве основы для пользовательского кода:

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

Или его тип-наследник (появившийся только в C# 2.0 вместе с обобщениями):

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

При этом предполагается, что при возникновении события будет передаваться:

- Ссылка на издателя – `sender` (может быть `null`, если событие статическое);
- Типизированный объект [EventArgs](#) или любой из его наследников для передачи данных о событии (которые могут быть [EventArgs.Empty](#), если необходимость в передаче данных отсутствует).

# Пример: Класс-Таймер. Часть 1

```
using System;
using System.Timers;

public class MyTimerClass {
    public event EventHandler<ElapsedEventArgs> Elapsed; // Событие нужного типа.
    private Timer _measureTimer; // Объект-таймер для подсчёта времени.
    private void OnOneSecond(object obj, ElapsedEventArgs e)
        => Elapsed?.Invoke(this, e);

    public MyTimerClass() {
        _measureTimer = new Timer();
        // Подписка на таймер, его запуск на 1 секунду:
        _measureTimer.Elapsed += OnOneSecond;
        _measureTimer.Interval = 1000;
        _measureTimer.Enabled = true;
    }
}
```

Событие вызовется при срабатывании библиотечного таймера.

Класс System.Timers.Timer использует свой тип параметров события.

# Пример: Класс-Таймер. Часть 2

```
using System;
```

```
public class ClassA  
{
```

```
    public void TimerHandlerA(object sender, ElapsedEventArgs elapsedArgs)  
        => Console.WriteLine($"Handler A: timer event received, time:" +  
                             $" {elapsedArgs.SignalTime}");
```

Сигнатура экземплярного метода соответствует делегату Elapsed.

```
public class ClassB  
{
```

```
    public static void TimerHandlerB(object sender, EventArgs elapsedArgs)  
        => Console.WriteLine($"Handler B: timer event received, time:" +  
                             $" {elapsedArgs.SignalTime}");
```

Сигнатура статического метода соответствует делегату Elapsed.

# Пример: Класс-Таймер. Часть 3

```
using System;
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
{
```

```
    MyTimerClass timer = new MyTimerClass();
```

```
    ClassA subscriberA = new ClassA();
```

```
    timer.Elapsed += subscriberA.TimerHandlerA;
```

```
    timer.Elapsed += ClassB.TimerHandlerB;
```

```
    // Таймер срабатывает каждую секунду.
```

```
    System.Threading.Thread.Sleep(2000);
```

```
    timer.Elapsed -= ClassB.TimerHandlerB;
```

```
    System.Threading.Thread.Sleep(1000);
```

```
}
```

```
}
```

## Вариант вывода:

Handler A: timer event received, time: 17-Jan-22 13:05:01

Handler B: timer event received, time: 17-Jan-22 13:05:01

Handler A: timer event received, time: 17-Jan-22 13:05:02

Handler B: timer event received, time: 17-Jan-22 13:05:02

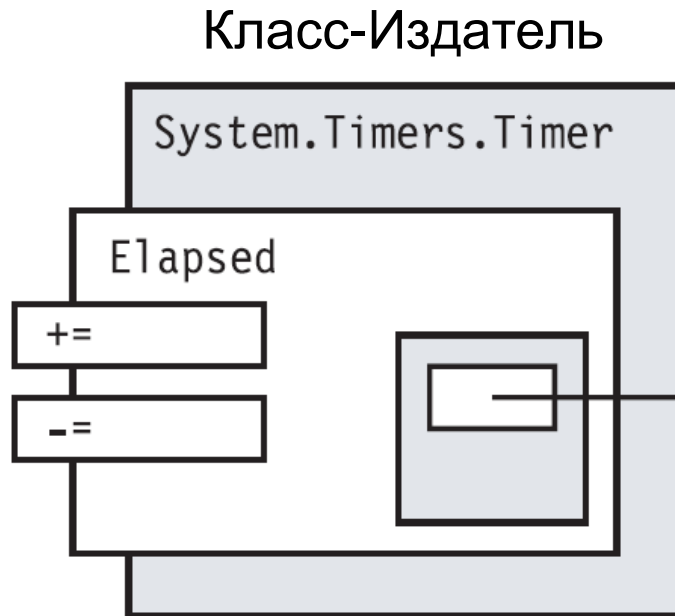
Handler A: timer event received, time: 17-Jan-22 13:05:03

Остановка выполнения основной программы на 2 секунды для демонстрации работы таймера.

Отписка статического метода класса B.

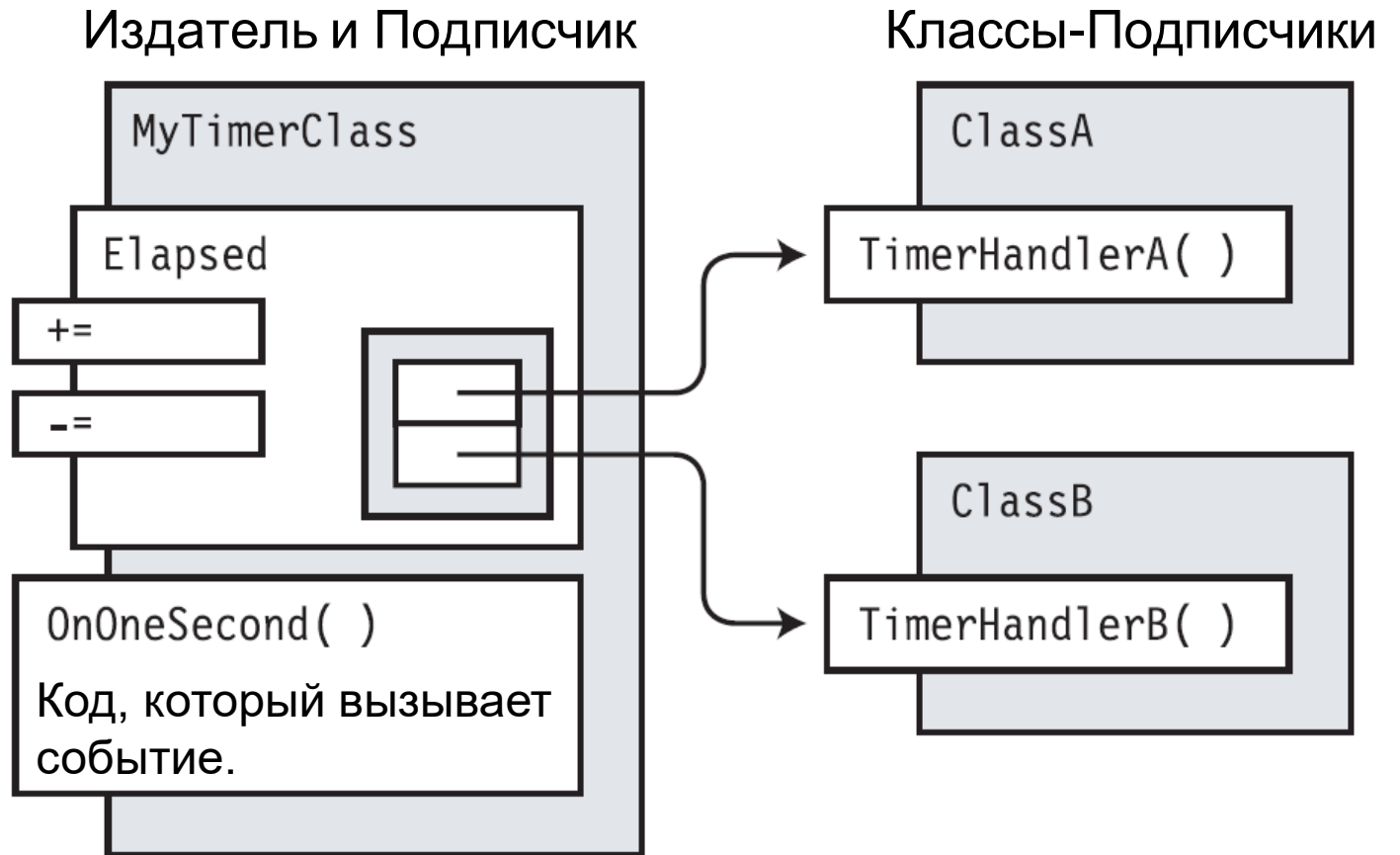
# Иллюстрация к Примеру с Таймером

Код Библиотеки .NET



В коде класса Timer имеется своё определение события Elapsed.

Наш Код





# Проектирование Типов для Шаблона Событий .NET

Для реализации стандартного шаблона генерации событий в .NET можно использовать 2 подхода:

- 1) (Устаревший вариант) Объявить пользовательский делегат-тип с явно указанным наследником EventArgs в качестве второго параметра, добавить событие этого типа в нужный класс:

```
public delegate void MyTimerEventHandler(object sender, MyTimerEventArgs e);  
+  
public event MyTimerEventHandler MyTimerEvent;
```

- 2) Использовать событие обобщённого делегат-типа EventHandler<T>:

```
public event EventHandler<MyTimerEventHandler> MyTimerEvent;
```

**На заметку:** большинство методов BCL использует именно первый вариант, т. к. в момент изначального написания библиотеки обобщения не поддерживались, а менять API не стали в целях сохранения обратной совместимости.

# Шаги Использования Шаблона Событий .NET

- 1) Объявить класс для передачи информации о событии – наследник EventArgs;
- 2) Выбрать делегат-тип для события:
  - a. Объявить делегат-тип с наследником EventArgs из п. 1 в качестве второго параметра;
  - b. Использовать делегат-тип EventHandler<T>.
- 3) Объявить событие делегат-типа из п. 2 внутри типа-издателя;
- 4) Добавить в тип-издатель метод генерации события (как правило, с именем **On<ИмяСобытия>**) и организовать передачу аргументов событию в нём.  
**На заметку:** объявление такого метода как **protected virtual** позволяет вызывать событие в наследниках и переопределять логику вызова;
- 5) Добавить в тип-издатель код, ответственный за генерацию события и вызов метода из п. 4;
- 6) Объявить в типе-подписчике метод обработчик события, соответствующий сигнатуре делегата из п. 2;
- 7) Добавить подписчика с помощью += к событию из п. 3.

# Пример: Банковский Счёт. Шаг 1

```
using System;
```

```
// Класс для передачи данных об изменении на счете:
```

```
public class BankEventArgs : EventArgs
```

```
{
```

```
    public decimal PreviousBalance { get; init; }
```

```
    public decimal NewBalance { get; init; }
```

```
    public BankEventArgs(decimal oldBalance, decimal newBalance)
```

```
        => (PreviousBalance, NewBalance) = (oldBalance, newBalance);
```

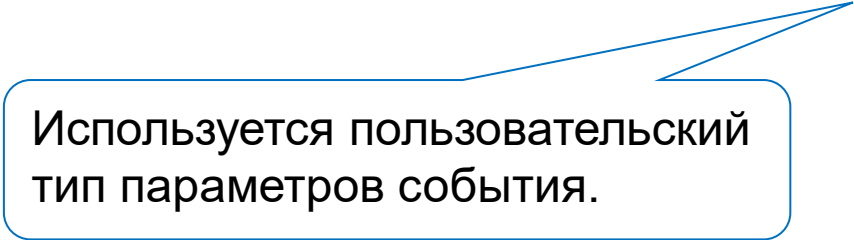
```
}
```

Наследование от EventArgs  
в соответствии шаблону.

# Пример: Банковский Счёт. Шаг 2 (Опционально)

// Делегат, соответствующий типу информации о событии:

```
public delegate void BankAccountEventHandler(object sender, BankEventArgs e);
```



Используется пользовательский тип параметров события.

# Пример: Банковский Счёт. Шаги 3-5

```
public class BankAccount {  
    public string Owner { get; private set; }  
    public decimal Balance { get; private set; }  
    public event EventHandler<BankEventArgs> AccountBalanceChanged;
```

**Шаг 3:** делегат-тип в соответствии с пунктом 2.b).

```
    public BankAccount(string ownerName, decimal initialBalance)  
        => (Owner, Balance) = (ownerName, initialBalance);  
    protected virtual void OnAccountBalanceChanged(object sender, BankEventArgs args)  
        => AccountBalanceChanged?.Invoke(sender, args);
```

**Шаг 4:** Метод On<Имя\_События>, контролирует вызов события.

```
    public void AddToAccount(decimal value) {  
        decimal oldBalance = Balance;  
        Balance += value;  
        OnAccountBalanceChanged(this, new(oldBalance, Balance));  
    }  
    public void RemoveFromAccount(decimal value) => AddToAccount(-value);
```

**Шаг 5:** Вызов генерации события.

```
}
```

# Пример: Банковский Счёт. Шаг 6

// Класс человек - владелец банковского счёта:

```
public class Person
```

```
{
```

```
    public string Name { get; private set; }
```

```
    public Person(string name) => Name = name;
```

Метод-обработчик соответствует  
сигнатуре делегат-типа  
EventHandler<BankEventArgs>.

```
    public void OnAccountBalanceChangedEventHandler(object sender,  
                                                    BankEventArgs args)
```

```
{
```

```
    Console.WriteLine($"{Name}: Old balance - {args.PreviousBalance:F3}, " +  
                        $" New balance - {args.NewBalance:F3}");
```

```
}
```

```
}
```

# Пример: Банковский Счёт. Шаг 7

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        Person person = new Person("Victor");
```

```
        BankAccount account = new BankAccount(person.Name, 50_000);
```

```
        account.AccountBalanceChanged += person.OnAccountBalanceChangedEventHandler;
```

```
        account.AddToAccount(2_000);
```

```
        account.RemoveFromAccount(25_000);
```

```
    }
```

```
}
```

Подписка человека на событие  
изменения банковского счёта.

## Вывод:

Victor: Old balance - 50000.000, New balance - 52000.000

Victor: Old balance - 52000.000, New balance - 27000.000

# Внутреннее Устройство Событий

Компилятор преобразует все события в несколько связанных конструкций:

- Закрытое поле указанного делегат-типа;
- **Метод доступа add** – для добавления подписчиков делегата;
- **Метод доступа remove** – для удаления подписчиков делегата.

**Обратите внимание:** для генерируемых компилятором add/remove организуются дополнительные меры по обеспечению потокобезопасности.

C# допускает явное определение add/remove, хотя, как правило, необходимость в этом возникает редко – для абстрактных событий часто достаточно реализации со стороны компилятора.

Однако, необходимость может возникнуть **для интерфейсов и виртуальных событий** (далее – подробнее об этом).



# Пример: Явная Реализация Методов Доступа

```
public class MyStringEventArgs : EventArgs
{
    public string MyString { get; set; }
    public MyStringEventArgs(string str) { MyString = str; }
}

public class EventPublisher {
    private EventHandler<MyStringEventArgs> _SmthHappened;

    public event EventHandler<MyStringEventArgs> SmthHappened {
        add {
            _SmthHappened += value;
            Console.WriteLine("_SmthHappened += value;");
        }
        remove {
            _SmthHappened -= value;
            Console.WriteLine("_SmthHappened -= value;");
        }
    }
    // Остальной код класса-издателя...
}
```

# Реализация Событий Компилятором

```
class EventDemo
{
    public event EventHandler Event;
}
```

Преобразуется в...



```
class EventDemo {
    private EventHandler _event;
    private object _objectLock = new object();
    public event EventHandler Event
    {
        add
        {
            lock (_objectLock)
            {
                _event += value;
            }
        }
        remove
        {
            lock(_objectLock)
            {
                _event -= value;
            }
        }
    }
}
```

Упомянутая ранее поддержка  
многопоточности.

# Проблема Использования Виртуальных Событий

Документация Microsoft явно рекомендует не использовать виртуальные события.

Данная проблема связана с тем, что компилятор неявно генерирует скрытое поле делегат-типа **и для родителя** (с virtual), **и для наследника** (с override). Это приводит к разному поведению при обращении к событию из методов родителя/наследника (т. к. фактически сгенерированные поля делегат-типов не связаны друг с другом).

Единственное решение в данном случае – явная реализацию методов доступа add/remove как в классе, объявляющем событие, так и в его наследниках.

Подробнее о проблеме:

<https://habr.com/ru/company/pvs-studio/blog/315600/> - на русском

<https://pvs-studio.com/en/blog/posts/csharp/0453/> - на английском

# Ссылки с Источниками по Событиям

Обзорная информация по событиям: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/>

Ключевое слово event, допустимые модификаторы:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event>

Подписка на события; методы доступа add и remove, их явная реализация:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-subscribe-to-and-unsubscribe-from-events>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/add>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/remove>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-implement-custom-event-accessors>

Стандартный шаблон событий .NET:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines>

Вызов событий базового типа из типов наследников:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-raise-base-class-events-in-derived-classes>

Проблема использования виртуальных событий: <https://pvs-studio.com/en/blog/posts/csharp/0453/>

Реализация интерфейсов с событиями:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-implement-interface-events>