|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.10 Делегаты и события |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Объявление и использование делегатов 3](#_Toc301432178)

[1.1. Зачем нужны делегаты? 3](#_Toc301432179)

[1.2. Определение делегата 4](#_Toc301432180)

[1.3. Вызов делегата 7](#_Toc301432181)

[1.4. Групповые делегаты 8](#_Toc301432182)

[1.5. Демонстрация: Объявление, создание и использование делегатов 11](#_Toc301432183)

[1.6. Анонимные методы 12](#_Toc301432184)

[2. Урок 2: Использование лямбда-выражений 13](#_Toc301432185)

[2.1. Что такое лямбда-выражения? 13](#_Toc301432186)

[2.2. Определение лямбда-выражений 14](#_Toc301432187)

[2.3. Область действия переменной в лямбда-выражении 16](#_Toc301432188)

[2.4. Демонстрация: Работа с лямбда-выражениями 17](#_Toc301432189)

[3. Урок 3: Обработка событий 17](#_Toc301432190)

[3.1. Что такое событие? 18](#_Toc301432191)

[3.2. Определение события 18](#_Toc301432192)

[3.3. Использование события 19](#_Toc301432193)

[3.4. Рекомендации по использованию событий 20](#_Toc301432194)

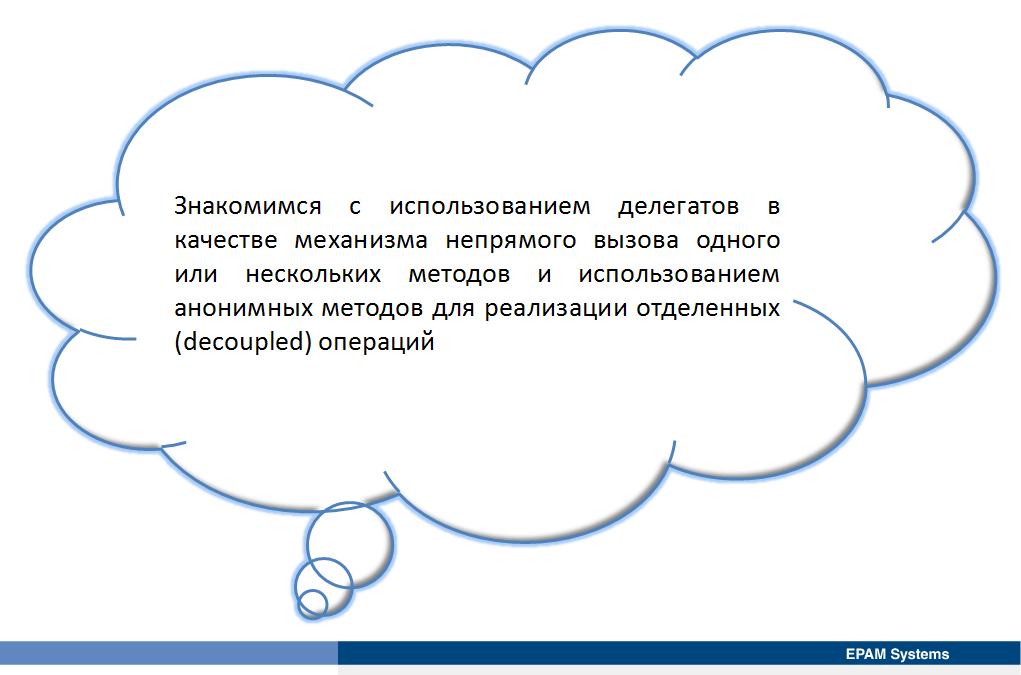
[3.5. Демонстрация: Использование событий 21](#_Toc301432195)

[3.6. Использование событий в графических приложений 22](#_Toc301432196)

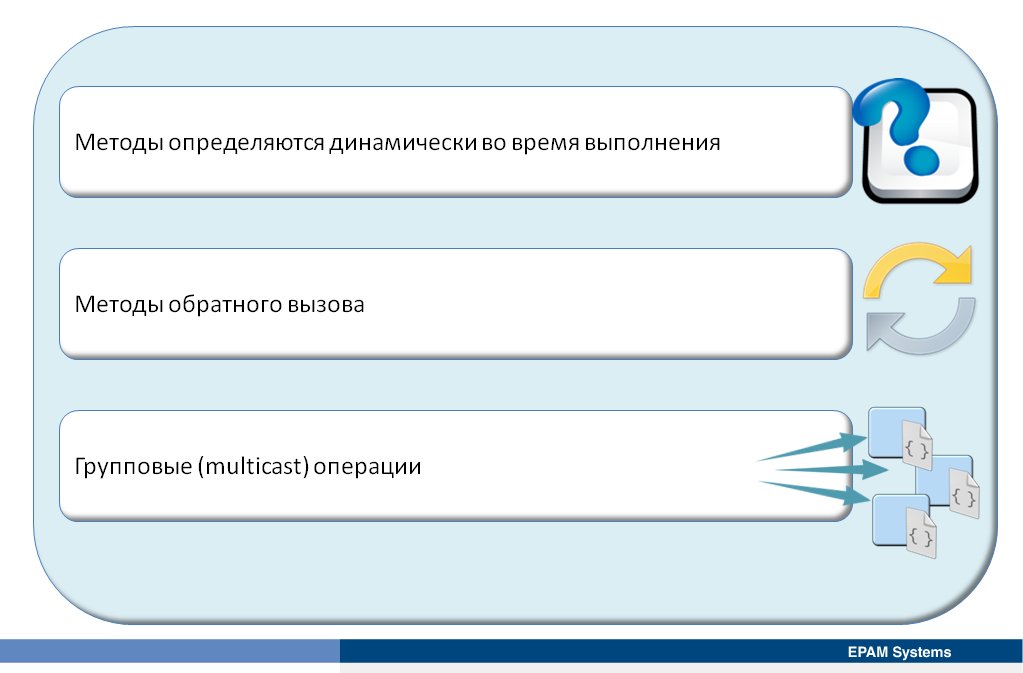
[3.7. Демонстрация: Использование событий в графических приложений 24](#_Toc301432197)

# Урок 1: Объявление и использование делегатов

Делегаты можно использовать для того, чтобы отделить функциональность от методов, которые ее реализуют. В уроке объясняется, как использовать делегаты в качестве реализации механизма обратного вызова (сallback methods) одного или нескольких методов и анонимные методы для реализации операций отделенных от метода.



## Зачем нужны делегаты?

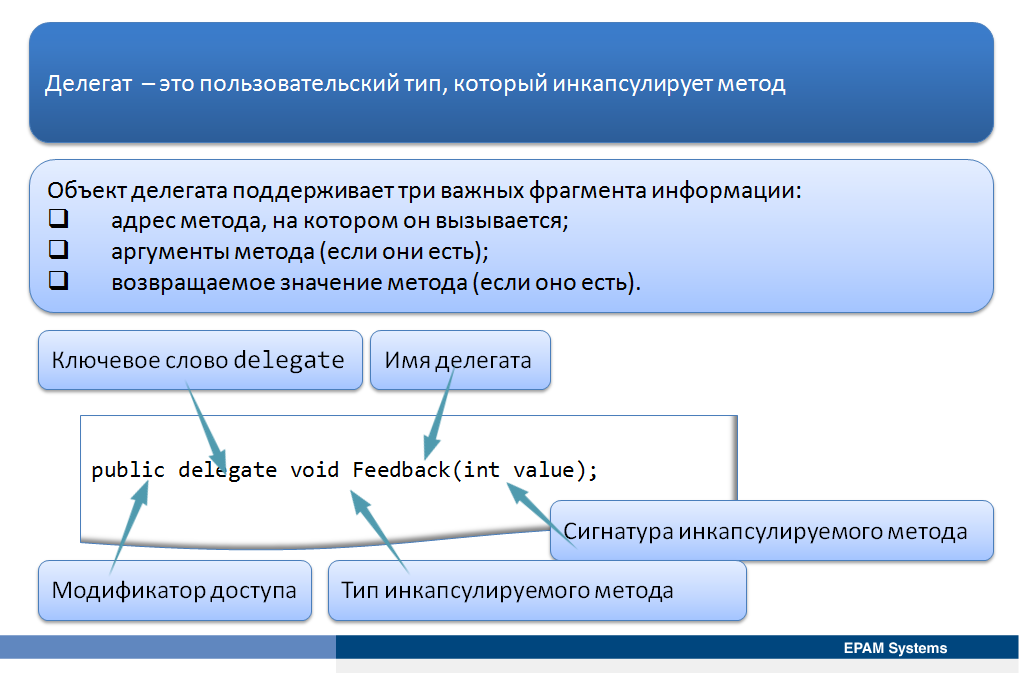


При разработке приложений с использованием Visual C# в большинстве случаев методы вызываются явно по имени. Функциональность и ее физическая реализация тесно связаны в методе. Однако, существуют сценарии, когда такой подход не годится. Например, при разработке фреймворка, в котором различные методы вызываются для выполнения определенной работы в зависимости от критериев, определяемых динамически при запуске приложения. Одним из способов реализации такой функциональности является использовании операторов if или switch, однако такой подход статичен и зависит от различных методов, доступных при написании кода. Более расширяемый подход заключается в использовании делегатов.

Другой сценарий касается методов обратного вызова. Сторонние производители, которые предоставляют сборки с методами, которые можно вызвать асинхронно, часто позволяют указать метод в коде пользователя, когда их метод завершен. При таком сценарии заранее неизвестны имена всех возможных в этом случае методов, поэтому эффективным решением является использование делегата, который можно связать с одним или более методов. Проще говоря, делегат ссылка на метод. Во время выполнения можно связать делегат с любым с методом, соответствующим его сигнатуре, и тогда для реализации функциональности метода, вызывается делегат. Как только делегат создан и снабжен необходимой информацией, он может динамически вызывать методы, на которые указывает. Каждый делегат в .NET Framework (включая специальные делегаты) автоматически снабжается способностью вызывать свои методы синхронно или асинхронно. Этот факт значительно упрощает задачи программирования, поскольку позволяет вызывать метод во вторичном потоке выполнения без создания и управления объектом Thread.

Следующими часто используемым для отделения функциональности от метода являются групповые (multicast) операции. Если операция является групповой, несколько методов могут реализовывать одну и ту же операцию. Можно добавить ссылки на все реализации методов делегату, а при вызове делегата, среда выполнения вызовет каждый метод по очереди. При использовании групповых операций реализующие их методы вызываются последовательно в соответствии с тем, как они были добавлены в делегат. Однако следует отметить, что если один из методов выбрасывает необработанное исключение, нет никакой гарантии, что приложение будет вызывать последующие методы.

## Определение делегата



Делегат – это пользовательский тип, который инкапсулирует метод. Объект типа делегат – это безопасный в отношении типов объект, указывающий на другой метод или, возможно, список методов приложения, которые могут быть вызваны позднее. Объект делегата поддерживает три важных фрагмента информации:

* • адрес метода, на котором он вызывается;
* • аргументы метода (если они есть);
* • возвращаемое значение метода (если оно есть).

В силу того, что делегат это ссылка на один или несколько методов, при определении делегата необходимо указать его сигнатуру. Имя делегата может быть любым. Методы, на которые ссылается делегат, должны иметь соответствующую делегату сигнатуру. Делегат определяется с помощью ключевого слова delegate. Как и для любого типа для делегата можно указать модификатор доступа,. При определении делегата создается новый тип, такой же как class или struct. Например,

public delegate void Feedback(int value);

При обработке данного типа делегат компилятор С# автоматически генерирует класс[[1]](#footnote-1), выглядящий примерно следующим образом

public class Feedback: MulticastDelegate

{

public Feedback(Object object, IntPtr method);

public virtual void Invoke(Int32 value);

public virtual IAsyncResult Beginlnvoke(Int32 value, AsyncCallback callback, Object object);

public virtual void EndInvoke(IAsyncResult result);

}

Сгенерированный компилятором класс Feedback определяет конструктор и три общедоступных метода (Рис. 1).

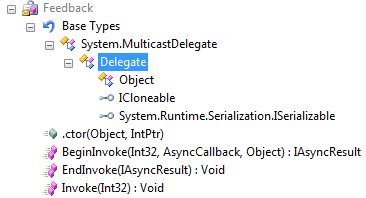


Рис. 1.

Конструктор такого класса принимает два параметра – ссылку на объект и целое число, ссылающееся на метод обратного вызова. Ссылка на объект передается в параметре object конструктора, а специальное значение IntPtr (получаемое из маркеров метаданных MethodDef или MethodRef), идентифицирующее метод, передается в параметре method. В случае статического метода в параметре object передается null. Внутри конструктора значения этих параметров сохраняются в закрытых полях \_target и \_methodPtr. Конструктор также заносит null в поле \_invocationList, используемое в групповых вызовах методов. Таким образом, любой объект делегата на деле является оболочкой метода и объекта, обрабатываемого этим методом. Поэтому при наличии двух следующих строк кода:

Feedback fbStatic = new Feedback(Program.FeedbackToConsole);

Feedback fbInstance = new Feedback(new Program.FeedbackToFile);

переменные fbStatic и fbInstance ссылаются на два разных инициализированных объекта-делегата Feedback (рис. 2).

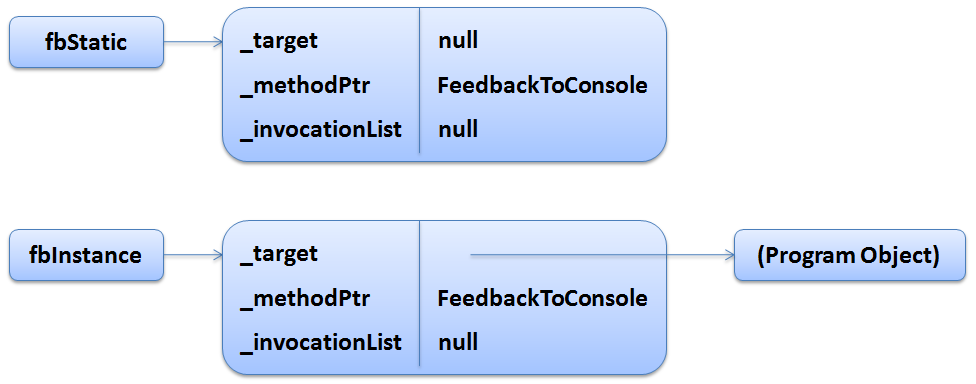


Рис. 2.

Метод Invoke используется для синхронного вызова каждого из методов, поддерживаемых объектом делегата; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Синхронный метод Invoke обычно явно в коде С# не вызываться, он вызывается «за кулисами» при применении соответствующего синтаксиса С#.

Методы BeginInvoke и EndInvoke предлагают возможность вызова текущего метода асинхронным образом, в отдельном потоке выполнения. Хотя в библиотеках базовых классов .NET предусмотрено целое пространство имен, посвященное многопоточному программированию (System.Threading), делегаты предлагают эту функциональность в готовом виде.

После объявления типа делегат можно объявить переменные этого типа:

Feedback F;

Переменные делегата инициализируются конкретными методами при использовании конструктора делегата с одним параметром – именем метода (или именем другого делегата). Если делегат инициализируется статическим методом, требуется указать имя класса и имя метода, для инициализации экземплярным методом указывается объект и имя метода. При этом метод должен обладать подходящей сигнатурой:

fbStatic = new Feedback(ClassName.SomeStaticFunction);

fbInstance = new Feedback(obj.SomeInstanceMethod);

Для инициализации делегата можно использовать упрощенный синтаксис – достаточно указать имя метода без применения new.

fbStatic = ClassName.SomeStaticFunction;

fbInstance = obj.SomeInstanceMethod;

Класс System.MulticastDelegate (в сочетании с его базовым классом System.Delegate) обеспечивает своих наследников доступом к списку, содержащему адреса методов, поддерживаемых типом делегата, а также несколькими дополнительными методами (и перегруженными операциями), чтобы взаимодействовать со списком вызовов. Ниже показаны некоторые методы System.MulticastDelegate и System.Delegate.

public abstract class MulticastDelegate : Delegate

{

// Возвращает список методов, на которые "указывает" делегат.

public sealed override Delegate[] GetInvocationList();

// Перегруженные операции.

public static bool operator ==(MulticastDelegate dl, MulticastDelegate d2);

public static bool operator != (MulticastDelegate dl, MulticastDelegate d2);

// Используются внутренне для управления списком методов, поддерживаемых делегатом.

private IntPtr \_invocationCount;

private object \_invocationList;

}

public abstract class Delegate : ICloneable, ISerializable

{

// Методы для взаимодействия со списком функций.

public static Delegate Combine(params Delegate[] delegates);

public static Delegate Combine(Delegate a, Delegate b);

public static Delegate Remove(Delegate source, Delegate value);

public static Delegate RemoveAll(Delegate source, Delegate value);

// Перегруженные операции.

public static bool operator ==(Delegate dl, Delegate d2);

public static bool operator !=(Delegate dl, Delegate d2);

// Свойства, показывающие цель делегата.

public MethodInfo Method { get; }

public object Target { get; }

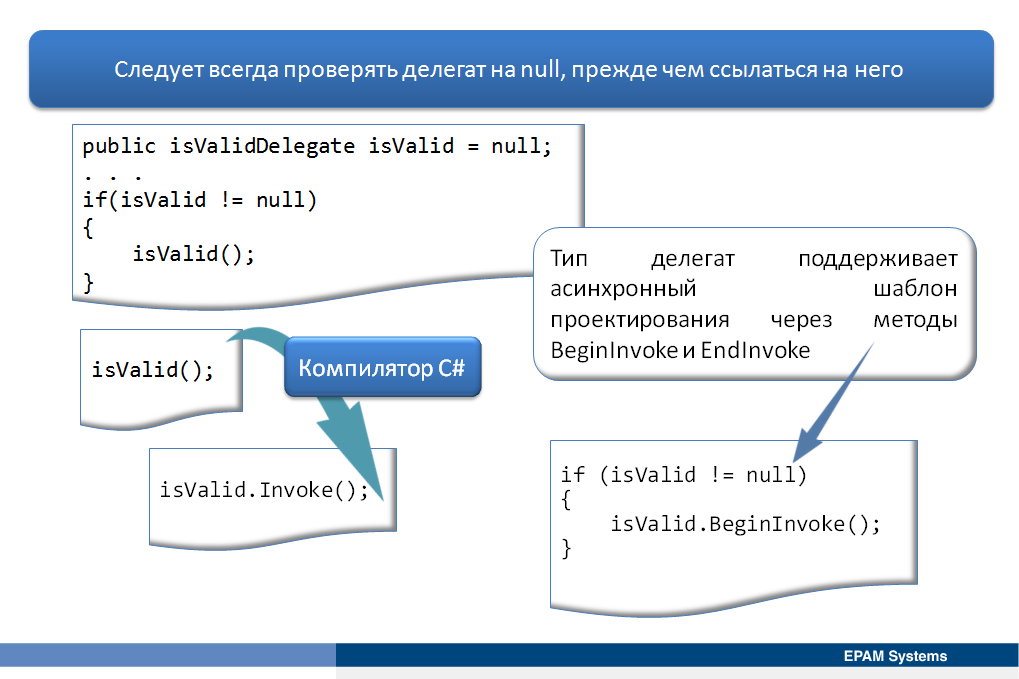
}

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| **Combine** | Статический метод добавляет метод в список, поддерживаемый делегатом. В С# этот метод вызывается за счет использования перегруженной операции += в качестве сокращенной нотации |
| **GetlnvokationListO** | Метод возвращает массив типов System.Delegate, каждый из которых представляет определенный метод, доступный для вызова |
| **Remove**  **RemoveAll** | Статические методы удаляют метод (или все методы) из списка вызовов делегата. В С# метод Remove может быть вызван неявно, посредством перегруженной операции -= |

В классе Delegate определены два неизменяемых открытых экземплярных свойства: Target и Method. При наличии ссылки на объект делегата можно запросить значения этих свойств. Target возвращает ссылку на объект, обрабатываемый при обратном вызове метода. В сущности, Target возвращает значение, хранимое в закрытом поле \_target. Если этот метод – статический, Target возвращает null. Свойство Method возвращает объект System.Reflection.Methodlnfo, описывающий метод обратного вызова (рис. 2). В сущности, у свойства Method есть внутренний механизм, который преобразует значение из закрытого поля \_methodPtr в объект Methodlnfo и возвращает его.

<http://go.microsoft.com/fwlink/?LinkId=192955>

## Вызов делегата



После определения делегата и создания его экземпляра делегат можно вызвать в коде. Делегат можно вызвать аналогично методу, используя имя экземпляра делегата со следующими за ним в скобках параметрами. Существует одно важное различие между вызовом делегата и вызовом метода: делегат может не ссылаться на методы и поэтому может быть равен null. Следует всегда проверять делегат на null, прежде чем ссылаться на него. Когда делегат не ссылается на какие-либо методы, он будет обращаться в null, когда делегат имеет ссылки на один или несколько методов, он не равен null. В следующем примере показан вызов делегата isValid.

public isValidDelegate isValid = null;

. . .

if(isValid != null)

{

isValid();

}

Следует отметить, что обнаружив строку вида

isValid();

компилятор генерирует код аналогичный, как если бы он встретил в исходном тексте следующее:

isValid.Invoke();

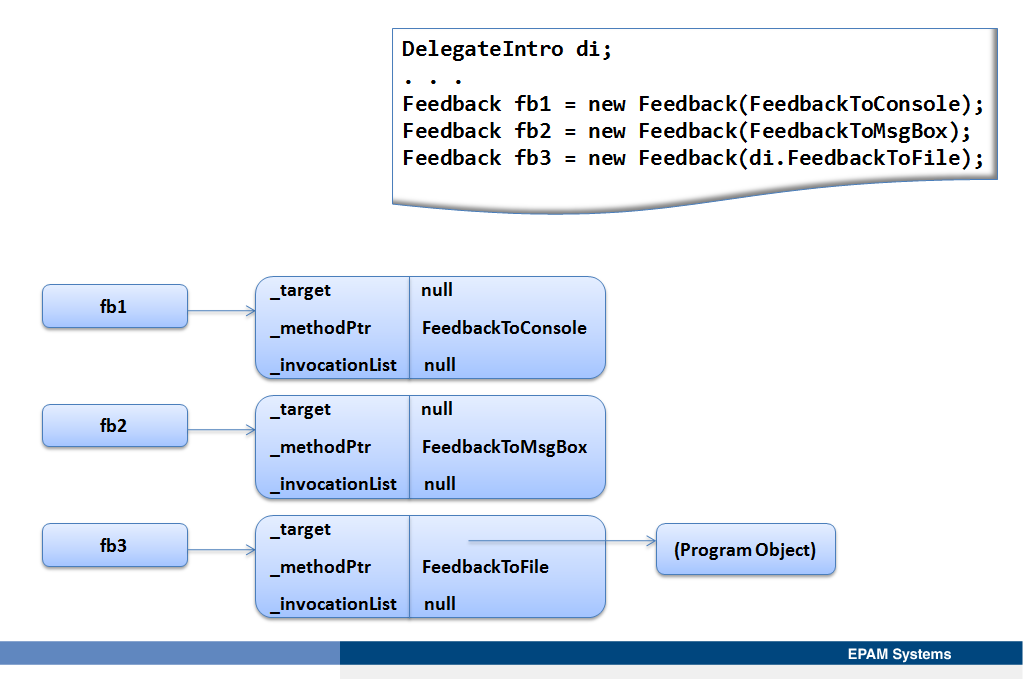
При вызове метода Invoke используются поля \_target и \_methodPtr для вызова желаемого метода на заданном объекте, при этом, сигнатура метода Invoke соответствует сигнатуре делегата.

При вызове делегата синхронно в случае группового вызова, каждый из инкапсулировнных в него методов вызывается по порядку. Если делегат имеет тип возвращаемого значения, значение из последнего метода реализации возвращается вызывающему приложению. Возвращаемые значения из любой другой реализации методов игнорируются.

В дополнение к вызову делегата синхронно, его можно вызвать асинхронно. Если нужно вызвать делегат асинхронно, он может ссылаться только на один метод, на групповой делегат ссылаться асинхронно нельзя. Тип делегат поддерживает асинхронный шаблон проектирования через методы BeginInvoke и EndInvoke. Приложение может вызвать метод BeginInvoke для запуска метода, на который ссылается делегат, асинхронно и использовать метод EndInvoke, чтобы получить любые данные, возвращаемые делегированным методом.

<http://go.microsoft.com/fwlink/?LinkId=192956>

## Групповые делегаты



Делегаты сами по себе невероятно полезны, но поддержка цепочек делает их еще полезнее. Цепочка делегатов (chaining) – это подмножество или набор объектов-делегатов, которое позволяет вызывать все методы, представленные делегатами набора. Для лучшего понимаяния принципа работы цепочки создаются три объекта типа делегат Feedback, на которые соответственно ссылаются три переменные fbl,fb2 и fb3 (Рис. 3).

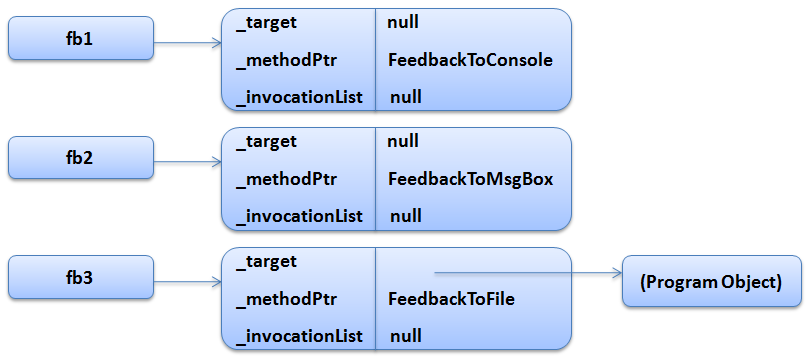
DelegateIntro di;

. . .

Feedback fb1 = new Feedback(FeedbackToConsole);

Feedback fb2 = new Feedback(FeedbackToMsgBox);

Feedback fb3 = new Feedback(di.FeedbackToFile);



Puc. 3.

Переменная-ссылка на объект-делегат Feedback должна ссылаться на цепочку или набор делегатов, служащих оболочками методам обратного вызова. Инициализация fbChain значением null говорит об отсутствии методов обратного вызова. Открытый статический метод Combine класса Delegate используется для добавления делегата в цепочку:

Feedback fbChain = null;

fbChain = (Feedback)Delegate.Combine(fbChain, fb1);

При выполнении последней строки кода метод Combine «видит» попытку объединить null и fb1. Код метода Combine просто возвращает значение fb1, а в переменной fbChain размещается ссылка на тот же объект-делегат, на который ссылается fb1 (Рис. 4).

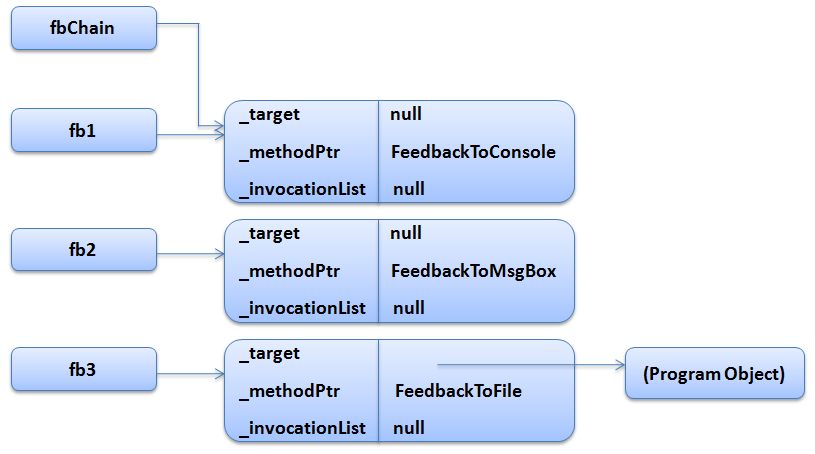
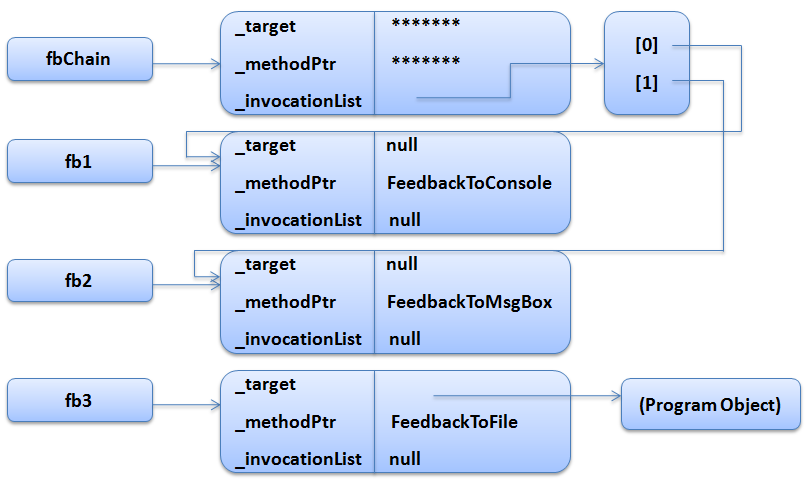


Рис. 4.

Чтобы добавить в цепочку еще один делегат, снова вызывается метод Combine.

fbChain = (Feedback)Delegate.Combine(fbChain, fb2);

Код метода Combine «видит», что fbChain уже ссылается на объект-делегат, и поэтому создает новый объект-делегат. Новый объект-делегат инициализирует свои закрытые поля \_target и \_methodPtr. Присваиваемые полям значения для данного обсуждения не важны, но важно то, что поле \_invocationList инициализируется ссылкой на массив объектов-делегатов. Первый элемент массива (с индексом 0) инициализируется ссылкой на делегат, служащий оберткой метода FeedbackToConsole (это делегат, на который сейчас ссылается fbChain). Второй элемент массива (индекс 1) инициализируется ссылкой на делегат, служащий оберткой метода FeedbackToMsgBox (на этот делегат ссылается fb2). Наконец, переменной fbChain присваивается ссылка на вновь созданный объект-делегат (Рис. 5).



Puc. 5.

Для создания третьего делегата снова вызывается метод Combine.

fbChain = (Feedback)Delegate.Combine(fbChain, fb3);

И снова, видя, что fbChain уже ссылается на объект-делегат, Combine создает новый объект-делегат (Рис. 6). Как и раньше, новый объект-делегат инициализирует свои закрытые поля \_target и \_methodPtr какими-то значениями, а поле \_invocationList инициализируется ссылкой на массив объектов-делегатов. Первый и второй элементы массива (индексы 0 и 1) инициализируются ссылками на те же делегаты, на которые ссылался предыдущий объект-делегат в массиве. Третий элемент массива (индекс 2) инициализируется ссылкой на делегат, служащий оберткой метода FeedbackToFile (на этот делегат ссылается fb3). В заключение, переменной fbChain присваивается ссылка на вновь созданный объект-делегат. Ранее созданный делегат и массив, на который ссылается его же поле \_invocationList, теперь подлежат обработке механизмом сборки мусора.

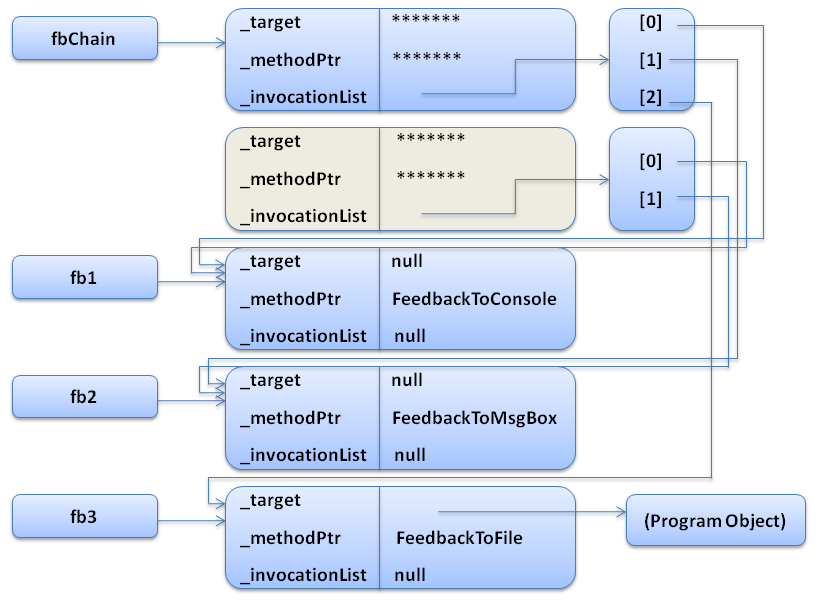


Рис. 6.

После выполнения всего кода создания цепочки, переменная fbChain передается методу Counter.

Counter(1, 2, fbChain);

. . .

private static void Counter(int from, int to, Feedback fb)

{

for (int val = from; val <= to; val++)

{

// If any callbacks are specified, call them

if (fb != null)

fb(val);

}

}

. . .

Внутри Counter содержится код, неявно вызывающий метод Invoke по отношению к объекту-делегату Feedback. Когда метод Invoke вызывается по отношению к делегату, на который ссылается fbCbain, делегат обнаруживает, что поле \_invocationList не равно null, и инициируется выполнение цикла, итерационно обрабатывающего все элементы массива путем вызова метода, оболочкой которого служит указанный делегат. Методы вызываются в следующей последовательности: FeedbackToConsole, FeedbackToMsgBox и FeedbackToFile.

Делегаты можно удалять из цепочки, вызывая статический метод Remove объекта Delegate. Например:

fbChain = (Feedback)Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));

Метод Remove просматривает массив делегатов (с конца и до члена с индексом 0), поддерживаемых объектом-делегатом, на который ссылается первый параметр (в примере fbChain). Remove ищет делегат, поля \_target и \_methodPtr которого совпадают с соответствующими полями второго параметра (в примере нового делегата Feedback). Если Remove находит совпадение и в массиве остается более одного элемента, создается новый объект-делегат – создается массив \_involocationList и инициализируется ссылкой на все элементы исходного массива за исключением удаляемого элемента – и возвращается ссылка на новый объект-делегат. При удалении последнего элемента в цепочке Remove возвращает null. Следует отметить, что за одтн раз Remove удаляет из цепочки лишь один делегат, а не все делегаты с заданными значениями в полях \_target и \_methodPtr.

Компилятор С# облегчает жизнь разработчикам, автоматически предоставляя перегрузку операций «+=» и «-=» для типов делегатов. Эти операции вызывают методы Delegate.Combine и DelegateRemove соответственно. Они упрощают построение груповых делегатов. С помощью перегруженных операций предыдущие операторы добавления и удаления методов из цеплчки можно записать в следующем виде.

DelegateIntro di;

. . .

Feedback fb1 = new Feedback(FeedbackToConsole);

Feedback fb2 = new Feedback(FeedbackToMsgBox);

Feedback fb3 = new Feedback(di.FeedbackToFile);

. . .

Feedback fbChain = null;

fbChain += fb1;

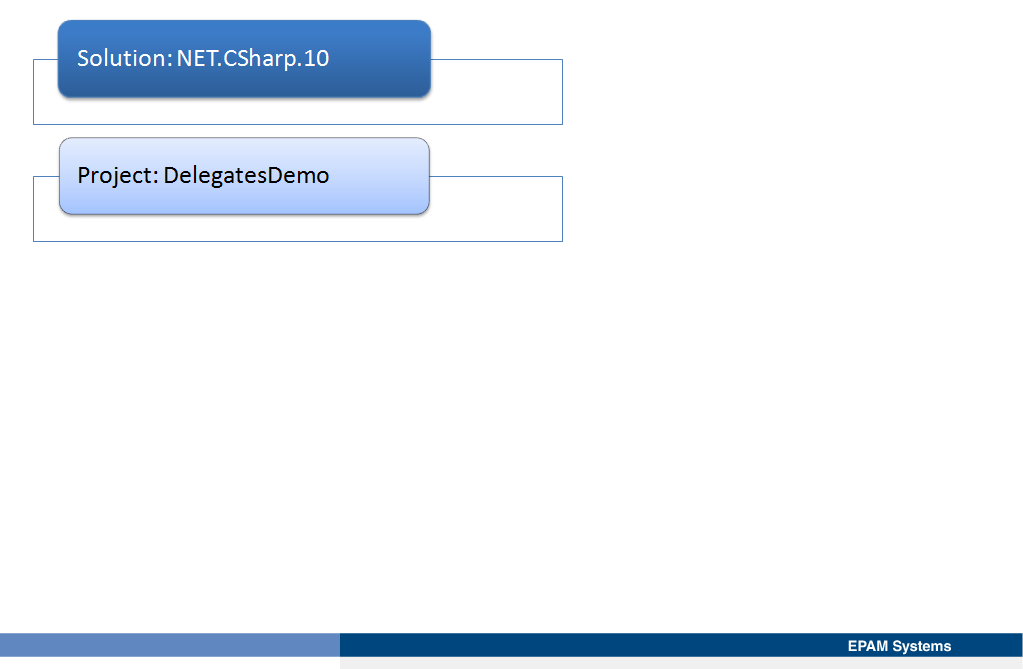
fbChain += fb2;

fbChain += fb3;

. . .

fbChain -= new Feedback(FeedbackToMsgBox);

## Демонстрация: Объявление, создание и использование делегатов



## Анонимные методы



Для использования делегата, он должен ссылаться на метод. Однако, часто бывает необходимо только отослать делегат к блоку кода, при этом создание метода, реализующего логику этого блока кода, добавляет накладные расходы (как минимум нужно придумать значимое имя метода). Для решения таких ситуаций Visual C# позволяет определить анонимные методы. Анонимным является методом, не имеющий имени, а имеющий только тип, список параметров и тело метода. Анонимный метод можно использовать с делегатом, вызывая его через делегат.

Для определения анонимного метода используется ключевое слово delegate, после которого можно дополнительно представить список параметров в круглых скобках. Если предоставляется список параметров, следует предоставить типы для этих параметров. И, наконец, предоставляется тело метода. Необходимо заключить тело метода в фигурные скобки, как в именованном методе. В следующем примере показано определение анонимного метода.

// Define a delegate type.

delegate int myDelegate(int number);

// Create an instance of the delegate type.

myDelegate myDelegateInstance = null;

public void addAnonymousMethodsToDelegate()

{

// Add an anonymous method to the delegate.

// Do not specify any parameters.

myDelegateInstance += new myDelegate(delegate

{

// Perform operation.

return 5;

});

// Add an anonymous method to the delegate.

// Specify parameters; the parameters must match

// the signature of the delegate.

myDelegateInstance += new myDelegate(delegate(int parameter)

{

return 10;

});

// Invoke the delegate.

int returnedValue = myDelegateInstance(2);

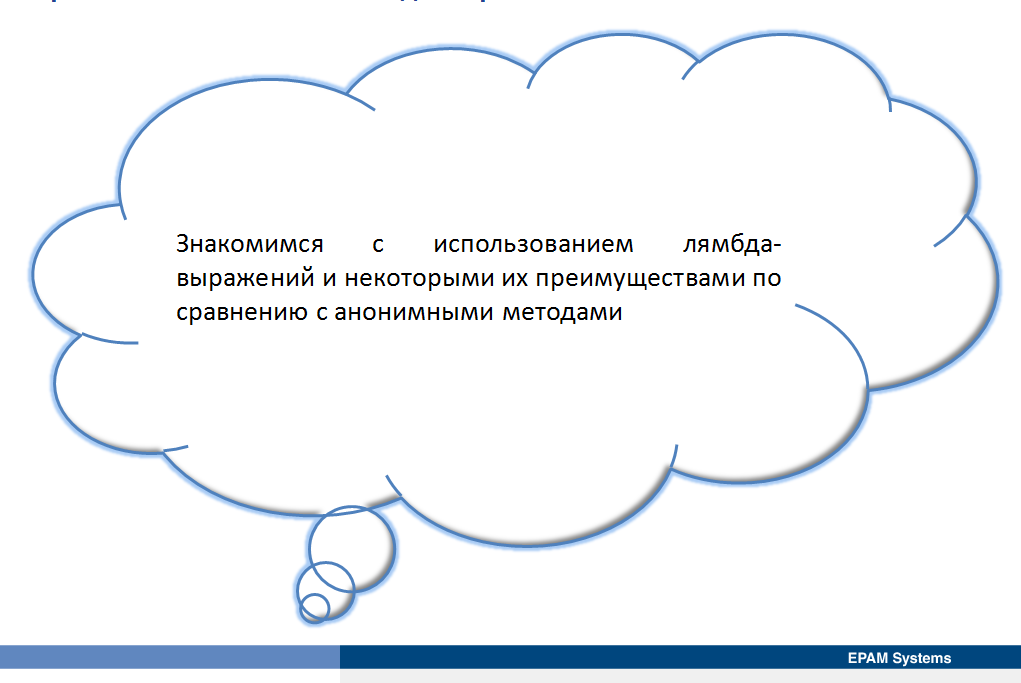
// returnedValue = 10 (as second method is called last).

}

Анонимные методы интересны тем, что могут обращаться к локальным переменным метода, в котором они определены. Формально такие переменные называются внешними (outer) переменными анонимного метода. Ниже отмечены некоторые важные моменты, касающиеся взаимодействия между контекстами анонимного и определяющего переменные методов.

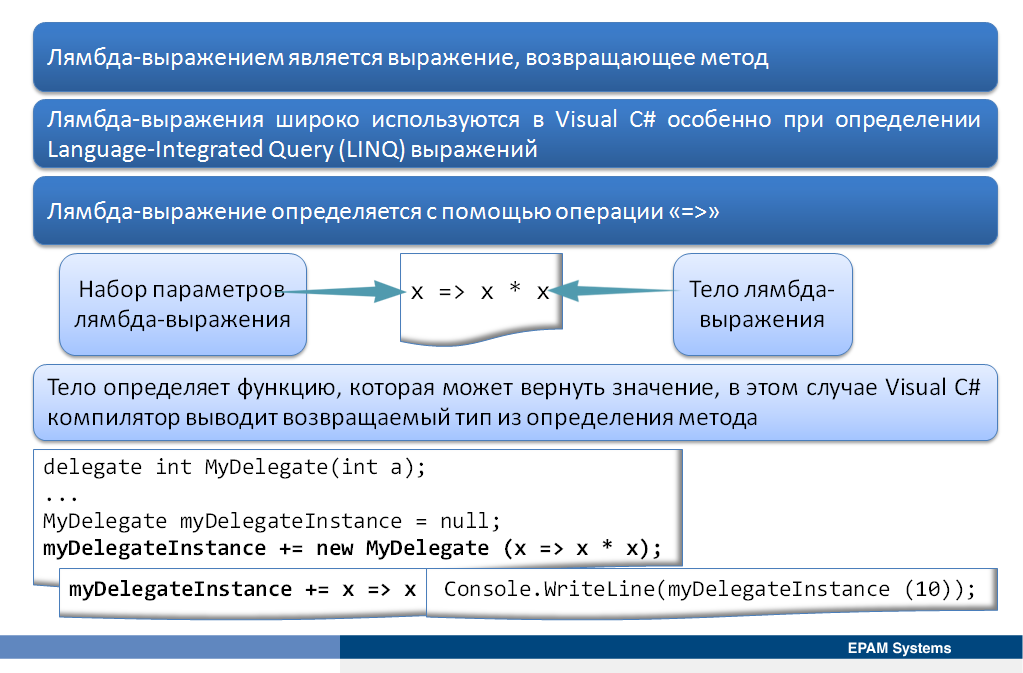
* Анонимный метод не имеет доступа к ref и out параметрам определяющего их метода.
* Анонимный метод не может иметь локальных переменных, имена которых совпадают с именами локальных переменных объемлющего метода.
* Анонимный метод может обращаться к переменным экземпляра (или статическим переменным) из контекста объемлющего класса.
* Анонимный метод может объявлять локальные переменные с теми же именами, что и у членов объемлющего класса (локальные переменные имеют отдельный контекст и скрывают внешние переменные-члены).

# Урок 2: Использование лямбда-выражений



Лямбда-выражения обеспечивают технику реализации анонимных методов, с помощью которой можно достичь более краткого кода, чем с делегатами. В уроке объясняется, как использовать лямбда-выражения и описываются некоторые преимущества лямбда-выражений по сравнению с анонимными методами.

## Что такое лямбда-выражения?



Лямбда-выражением является выражение, возвращающее метод. Эти выражения произошли из мира функционального программирования и особенно полезны для определения анонимных, но строго типизированных методов. У них есть смысловое математическое обоснование, и они широко используются в Visual C# особенно при определении Language-Integrated Query (LINQ) выражений.

Лямбда-выражение состоит из набора параметров и тела. Тело определяет функцию, которая может вернуть значение , в этом случае Visual C# компилятор выводит возвращаемый тип из определения метода.

Лямбда-выражение определяется с помощью операции =>.

x => x \* x

Прочитать это лямбда-выражение можно так «Для данного х, вычислить х \* х». Тип х не определен, он будет выведен компилятором при использовании лямбда-выражения. Кроме того будет выведен и тип возвращаемого значения.

На лямбда-выражение можно ссылаться из делегата.

delegate int MyDelegate(int a);

...

MyDelegate myDelegateInstance = null;

myDelegateInstance += new MyDelegate (x => x \* x);

В примере тип x и тип возвращаемого значения в лямбда-выражении определяются делегатом, принимающем целочисленный параметр и возвращающим целое число. Получить более естественный и краткий синтаксис можно сократив оператор, присваивающий лямбда-выражение myDelegateInstance, следующим образом.

delegate int MyDelegate(int a);

...

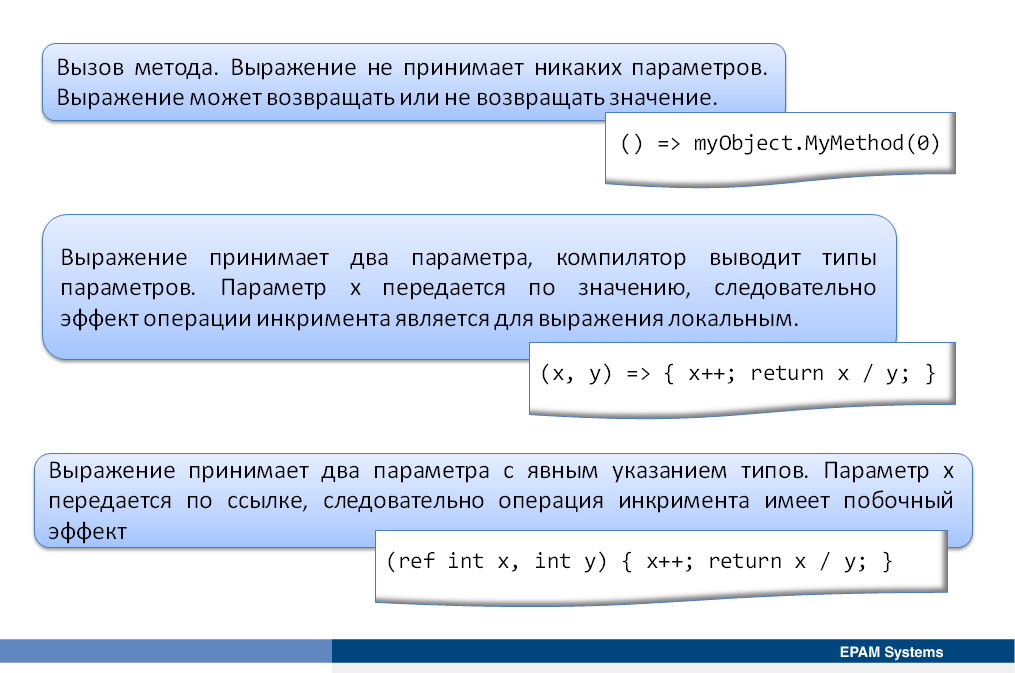
MyDelegate myDelegateInstance = null;

myDelegateInstance += x => x \* x;

При вызове делегата необходимо указать значения для всех параметров, которые принимает лямбда-выражение, и тело лямбда-выражения выполнится. Из тела лямбда-выражения можно вернуть любое возвращаемое значение так же, как при вызове обычного метода. В следующем примере показан вызов делегата на основе простого лямбда-выражения и делегата, описанного выше.

Console.WriteLine(myDelegateInstance (10)); // Displays the value 100

## Определение лямбда-выражений



Лямбда-выражения могут принимать множество различных форм. Лямбда-выражения были первоначально частью математической нотации, называемой лямбда-вычиления (Lambda Calculus), которая обеспечивала обозначения для описания функций. Хотя язык Visual C# расширил синтаксис и семантику лямбда-вычилений в осуществлении лямбда-выражений, многие из оригинальных математических принципов остались в силе.

Тело лямбда-выражения может быть простым выражением, как показано в предыдущем разделе, или это может быть блок кода Visual C#, который определяет тело метода, заключенное в фигурные скобки. При определении тела метода, можно использовать любые программные конструкции Visual C#.

Лямбда-выражение может принимать более одного параметра, в этом случае указывается список параметров, заключенных в круглые скобки.

delegate int AddDelegate(int a, int b)

...

AddDelegate myAddDelegate = null;

myAddDelegate += (x, y) => x + y;

Лямбда-выражения могут также принимать нулевые параметры. В этом случае указыватеся пустой список параметров. Следующие примеры кода показывают много различных форм лямбда-выражений доступных в Visual C#.

// A simple expression that returns the square of its parameter.

// The type of parameter x is inferred from the context.

x => x \* x

// Semantically the same as the preceding expression, but using a Visual C# statement

//block as a body rather than a simple expression.

x => { return x \* x ; }

// A simple expression that returns the value of the parameter divided by 2.

// The type of parameter x is stated explicitly.

(int x) => x / 2

// Calling a method. The expression takes no parameters.

// The expression may or may not return a value.

() => myObject.MyMethod(0)

// Multiple parameters; the compiler infers the parameter types.

// The parameter x is passed by value, so the effect of the ++ operation is

// local to the expression.

(x, y) => { x++; return x / y; }

// Multiple parameters with explicit types.

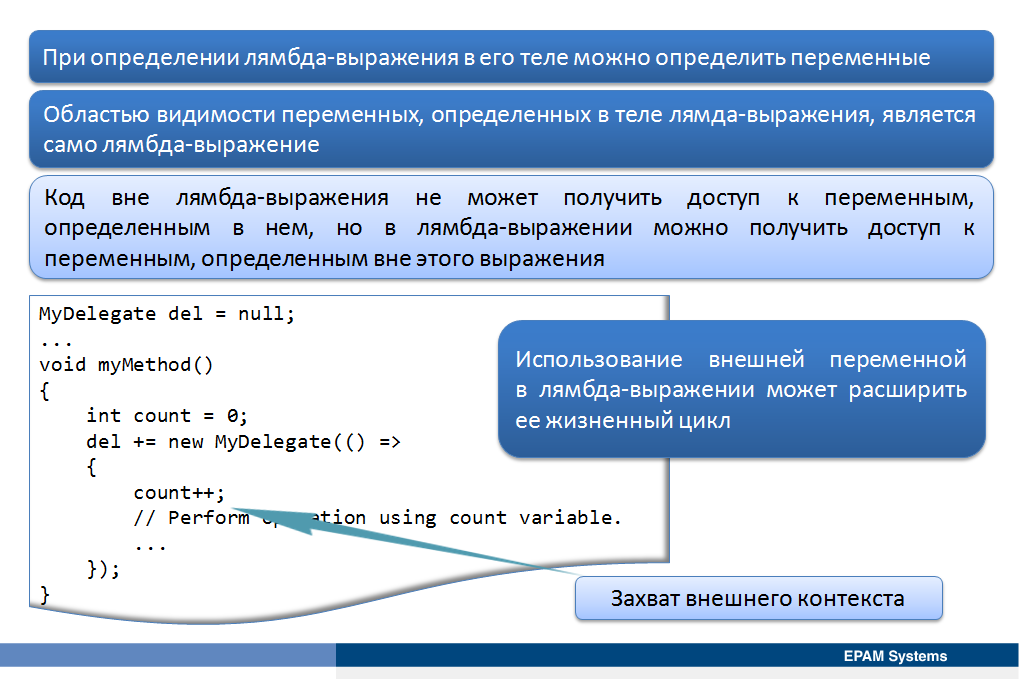
// Parameter x is passed by reference, so the effect of the ++ operation is permanent.

(ref int x, int y) { x++; return x / y; }

Хотя в лямбда-выражении возможно использовать любой код Visual C#, не считается хорошим стилем программирования определять код, изменяющий данные, используемые приложением в другом месте, это приводит с побочными эффектами, часто являющимся причиной ошибок, которые трудно отследить.

<http://go.microsoft.com/fwlink/?LinkId=192957>

## Область действия переменной в лямбда-выражении



При определении лямбда-выражения в его теле можно определить переменные. Областью видимости этих переменных является лямбда-выражение. Когда лямбда-выражение завершается, переменная выходит из области видимости. Код вне лямбда-выражения не может получить доступ к переменным, определенным в нем. Однако в лямбда-выражении можно получить доступ к переменным, определенным вне этого выражения. Однако время жизни этих переменных будет расширено до тех пор, пока лямбда-выражение само не выдет из области видимости, что может иметь влияние на сбор мусора.

Анонимные методы и лямбда-операторы способны захватывать внешний контекст вычисления. Если при описании тела анонимного метода применялась внешняя переменная, вызов метода будет использовать текущее значение переменной. Захват внешнего контекста иначе называют замыканием (closure). В следующем примере показано, как лямбда-выражение, которое присваивается делегату в методе, может использовать переменные, определеные в этом методе.

MyDelegate del = null;

...

void myMethod()

{

int count = 0;

del += new MyDelegate(() =>

{

count++;

// Perform operation using count variable.

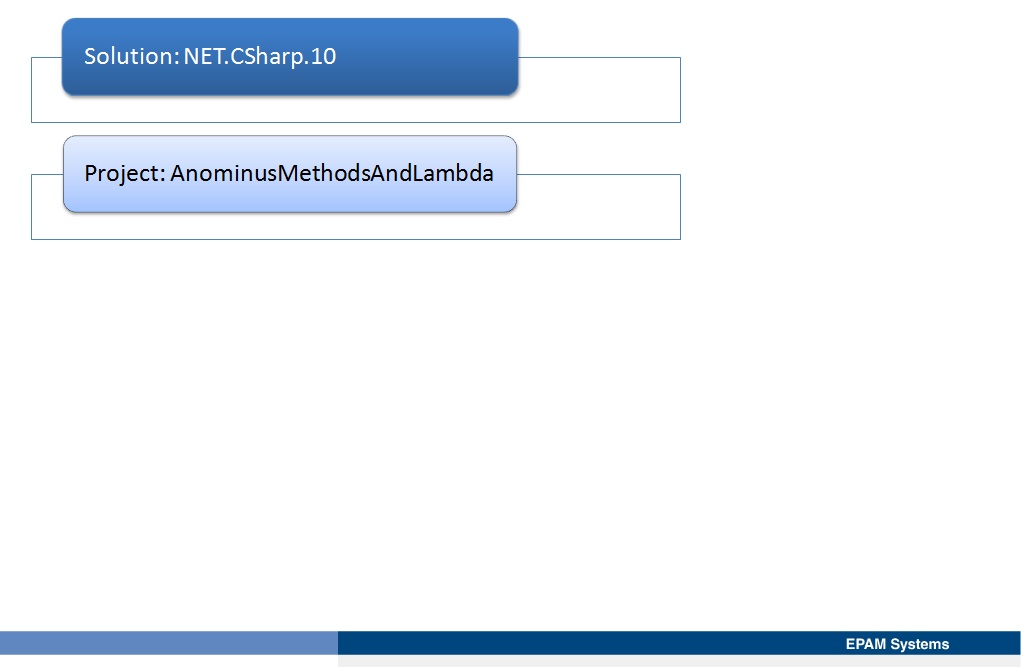
...

});

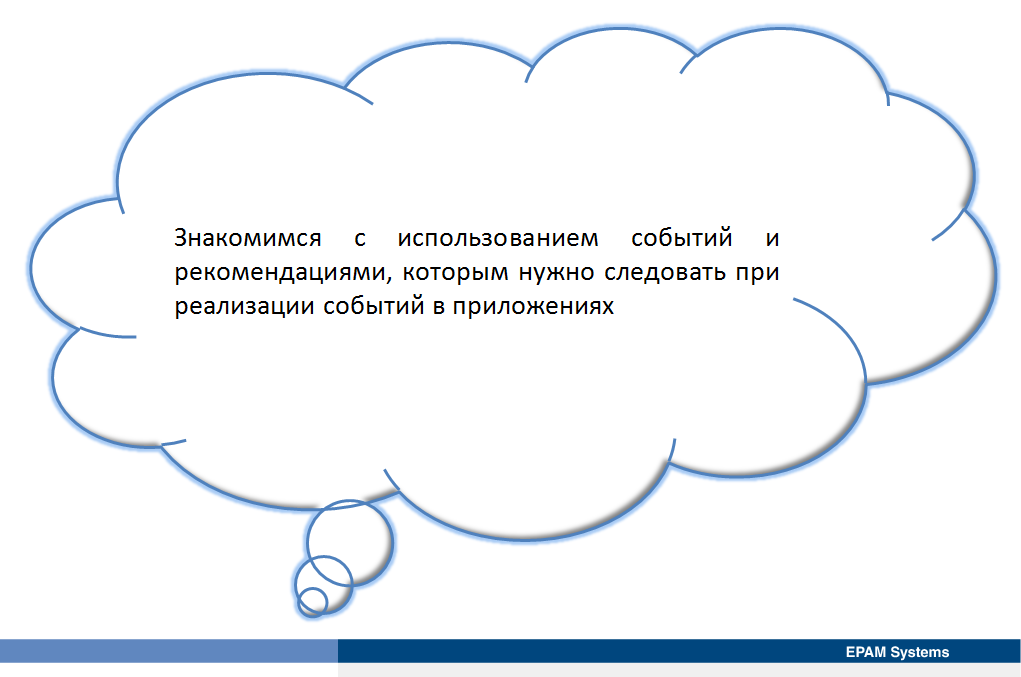
}

В предыдущем примере если какая-либо ссылка на делегат del останется существовать, когда метод завершится, переменная count останется. Только когда все ссылки на делегат del удалятся, сборщик мусора отметит переменную count для удаления. В данном примере, данные это целое поле, использующее очень мало ресурсов, однако, лямбда-выражение может использовать любой объект любого типа, находящийся в области видимости, когда выражение определено. И если объект большой и использует много ресурсов, это сказывается на производительности приложения. Всегда следует внимательно рассмотреть вопрос о необходимости ссылаться на переменные, определеные вне сферы действия лямбда-выражения.

## Демонстрация: Работа с лямбда-выражениями

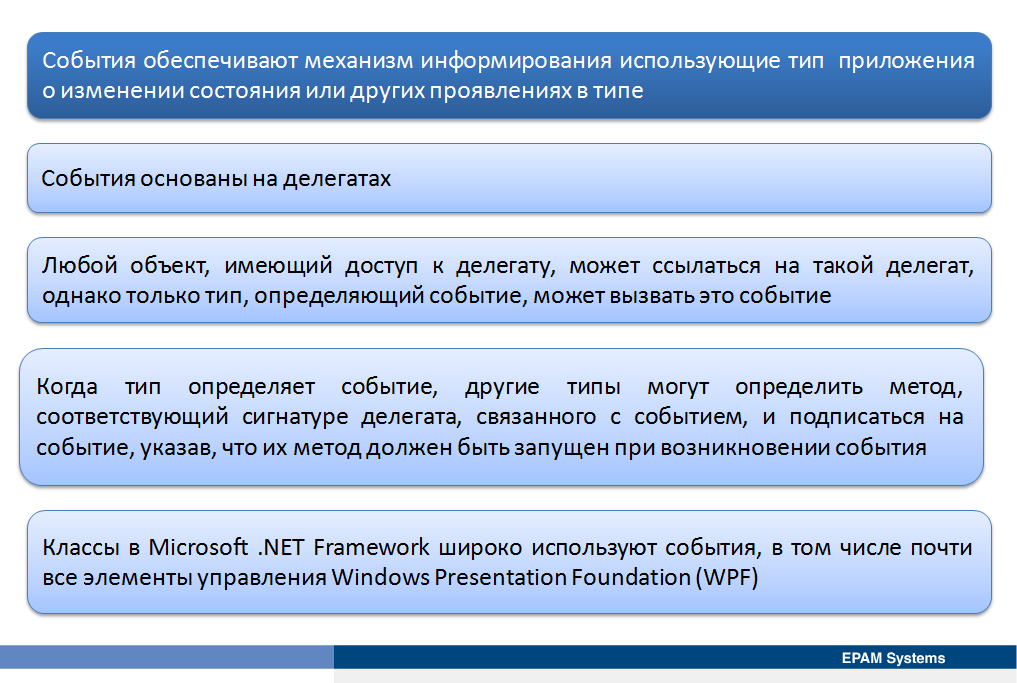


# Урок 3: Обработка событий



События позволяют указать, что в приложении произошло что-то значительное, и, возможно, другие элементы приложения должны быть проинформированы об этом событии. Например, когда пользователь нажимает кнопку в графическом приложении, он, как правило, хочет, чтобы запускался блок кода, выполняющий некоторые действия, связанные с нажатием кнопки. В уроке объясняется, как использовать события и приводятся некоторые рекомендации, которым нужно следовать при реализации событий в приложениях.

## Что такое событие?

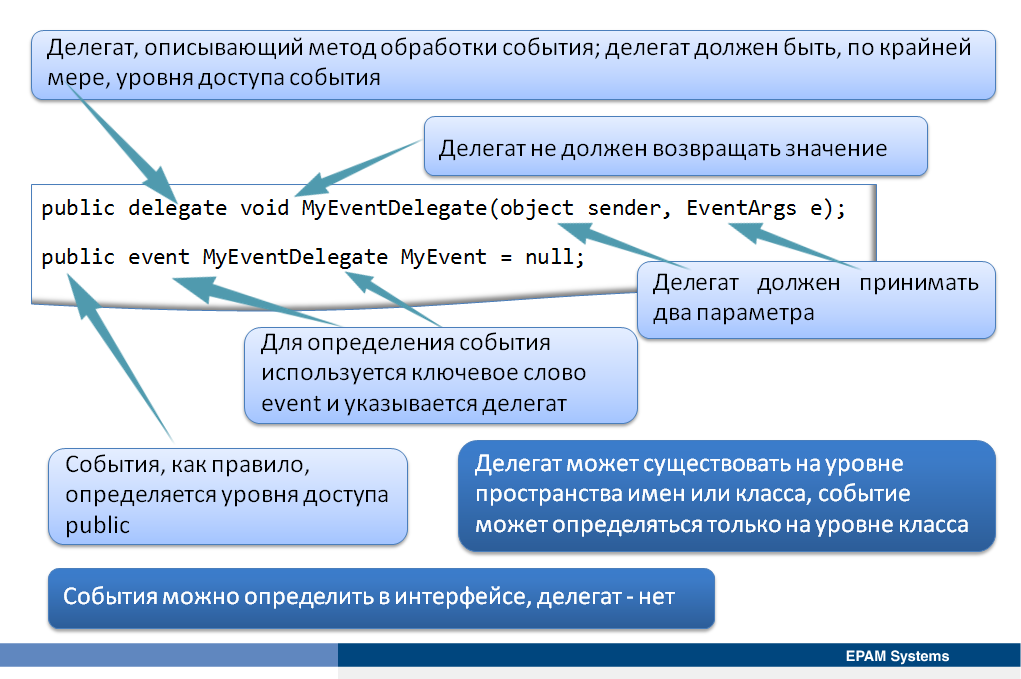


В Visual C# события очень похожи на делегаты. События действительно основаны на делегатах, хотя семантически, отличаются от них в назначении. Тип использует события для свидетельствования о значительном изменении в приложении и организации делегата для вызова обработчика события. В то время как любой объект, имеющий доступ к делегату, может ссылаться на такой делегат, только тип, определяющий событие, может вызвать это событие.

Когда тип определяет событие, другие типы могут определить метод, соответствующий сигнатуре делегата, связанного с событием. Эти типы могут подписаться на событие и указать, что их метод должен быть запущен при возникновении события.

Классы в Microsoft .NET Framework широко используют события, в том числе почти все элементы управления Windows Presentation Foundation (WPF). События следует использовать только тогда, когда разрабатывается тип, которому необходимо сообщать многим классам об изменении своего состояния.

## Определение события



События основаны на делегатах, поэтому прежде чем определять событие, следует определить делегат, описывающий метод обработки события. Существует соглашение, определяющее стандартную сигнатуру для делегата события, которому нужно, по мере возможности, стараться следовать. Согласно стандартному соглашению делегат события не должен возвращать значение и должен принимать два параметра. Первому параметру необходимо иметь тип object и имя sender. Второй параметр должен быть производным от класса EventArgs пространства имен System с именем e. Если событие не требует передачи аргументов события при его вызове, можно использовать класс EventArgs напрямую. В следующем примере показан типичный делегат для использования с событием.

public delegate void MyEventDelegate(object sender, EventArgs e);

Для определения события используется ключевое слово event и указывается делегат. Рекомендуется инициализировать событие значением null, поскольку это позволяет типу определить подписался ли какой-нибудь тип на событие до его возникновения (когда метод подписывается на событие, событие больше не будет null). В следующем примере показано определение события MyEvent на основе делегата MyEventDelegate.

public event MyEventDelegate MyEvent = null;

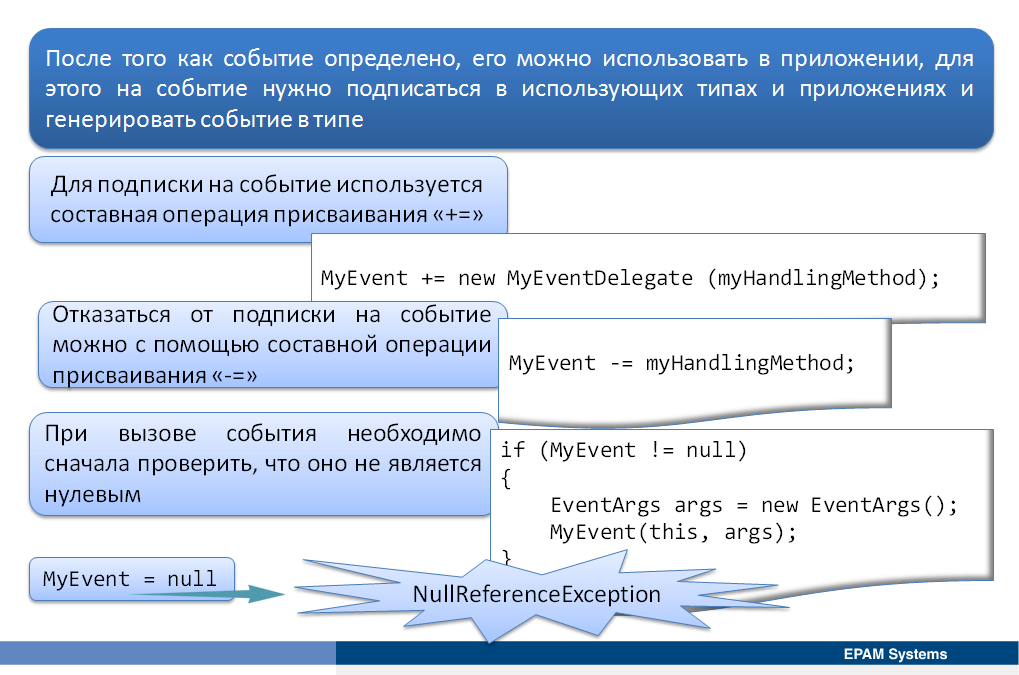
Делегат это тип, а, следовательно, может существовать на уровне пространства имен или класса, , событие же это член класса и может определяться только на уровне класса. Событие уровня класса может использовать делегат любого уровня, включая делегатов, существующих в других сборках. Обычной практикой, является использование некоторых из стандартных делегатов, определеных в .NET Framework.

При определении события, оно, как правило, имеет уровень доступа public для того, чтобы использующие тип классы могли иметь доступ к нему. При определении события, следует помнить, что делегат должен быть, по крайней мере, уровня доступа события в противном случае использующие тип классы не будут в состоянии создать экземпляры делегата и, следовательно, не смогут подписаться на событие.

В отличие от делегата, события можно определить в интерфейсе.

<http://go.microsoft.com/fwlink/?LinkId=192958>

## Использование события



После того как событие определено, его можно использовать в приложении. Для этого на событие нужно подписаться в использующих типах и приложениях и генерировать событие в типе.

Чтобы подписаться на событие используется составная операция присваивания «+=» точно так же, как при добавлении ссылки на метод делегату. В следующем примере показано добавление подписчика событию MyEvent.

MyEvent += new MyEventDelegate (myHandlingMethod);

В дополнение к подписке на событие можно отказаться от подписки на событие. Для этого используется составная операция присваивания «-=». В следующем примере показано удаление подписчика на событие MyEvent.

MyEvent -= myHandlingMethod;

Для генерации события используется имя события и указываются параметры в скобках, аналогично вызову метода.При вызове события необходимо сначала проверить, что оно не является нулевым. Если у события нет подписчиков, не следует пытаться его вызвать. При попытке вызвать событие, не имеющее подписчиков, код будет генерировать исключение NullReferenceException.

if (MyEvent != null)

{

EventArgs args = new EventArgs();

MyEvent(this, args);

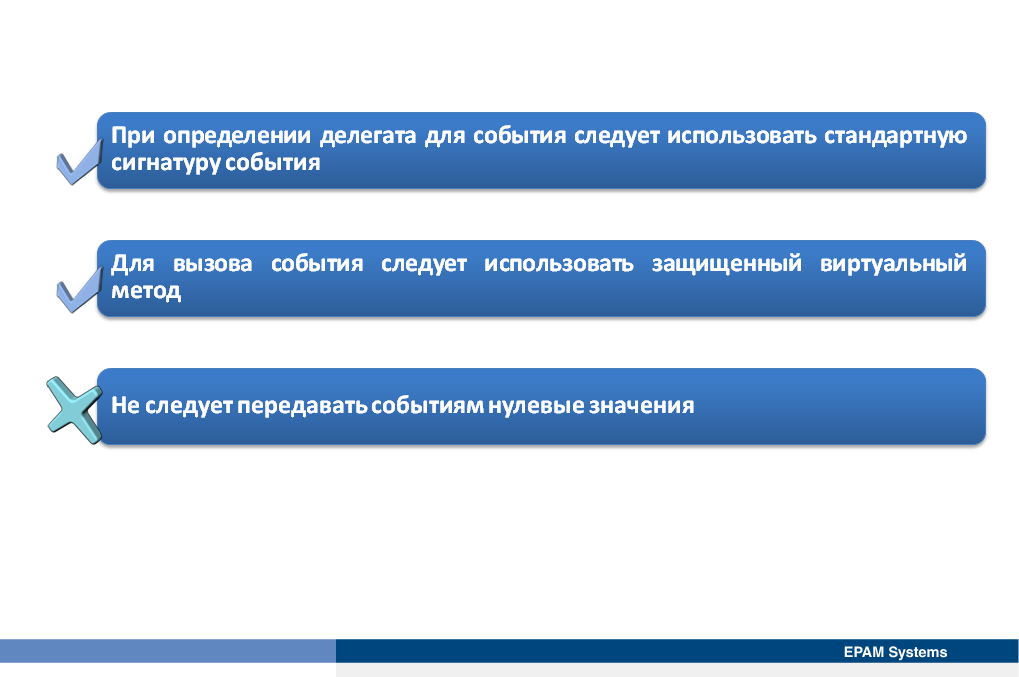
}

Метод инициализации события следует стараться именовать с префиксом On.

http://go.microsoft.com/fwlink/?LinkId=192959

http://go.microsoft.com/fwlink/?LinkId=192962

## Рекомендации по использованию событий



При разработке типа, использующего событие, следует придерживаться определенным «негласным правилам», что будет способствовать тому, что разработчикам, использующих данный тип, потребуются минимальные дополнительные знания для создания собственных типов. Наилучшая практика также помогает обеспечить, чтобы любые внесенные в будущем изменения не вызывали критических изменений в приложениях, уже использующих данный тип.

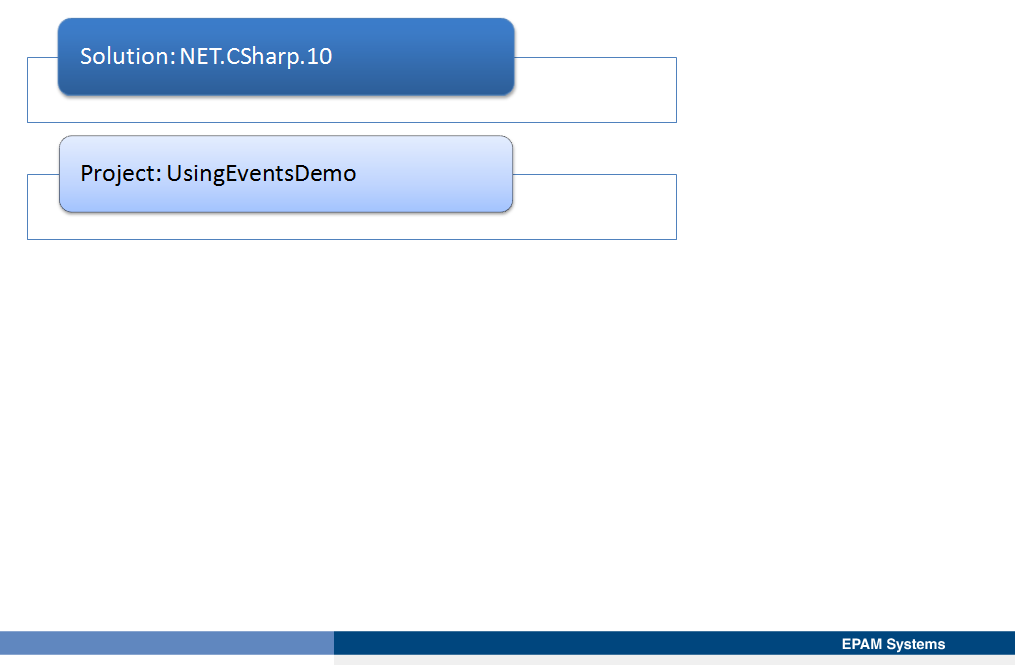
**Практический совет: при определении делегата для события следует использовать стандартную сигнатуру события.** Событие должно возвращать тип void в качестве возвращаемого значения и принимать два параметра: первый параметр должен быть object с именем sender, а второй параметр должен иметь тип производный от класса EventArgs с именем e. Вместо использования класса EventArgs рекомендуется определить собственный класс, наследуемый от класса EventArgs; класс EventArgs можно использовать, если никогда в будущем не надо будет передавать информацию в аргументах события. При разработке класса для хранения аргументов события для типа желательно, чтобы имя класса было MyNameEventArgs, где MyName либо название события, либо название, связанное с типом передаваемых аргументов.

**Практический совет: для вызова события следует использовать защищенный виртуальный метод.** При инициировании события в типе нужно определить защищенный виртуальный метод, генерирующий событие. Метод должен быть назван в соответствии с именем события и с префиксом On. Например, метод вызова события MyEvent будет называться OnMyEvent. On событию необходимо принимать такие же параметры, как событию, и содержать логику: например, прежде чем генерировать событие, проверять, не является ли оно нулевым. Если событие необходимо генерировать в типе, можно использовать On метод, а не генерировать метод непосредственно. Это уменьшает дублирование кода, поскольку не нужно проверять, является ли это событие нулевым каждый раз при его вызове.

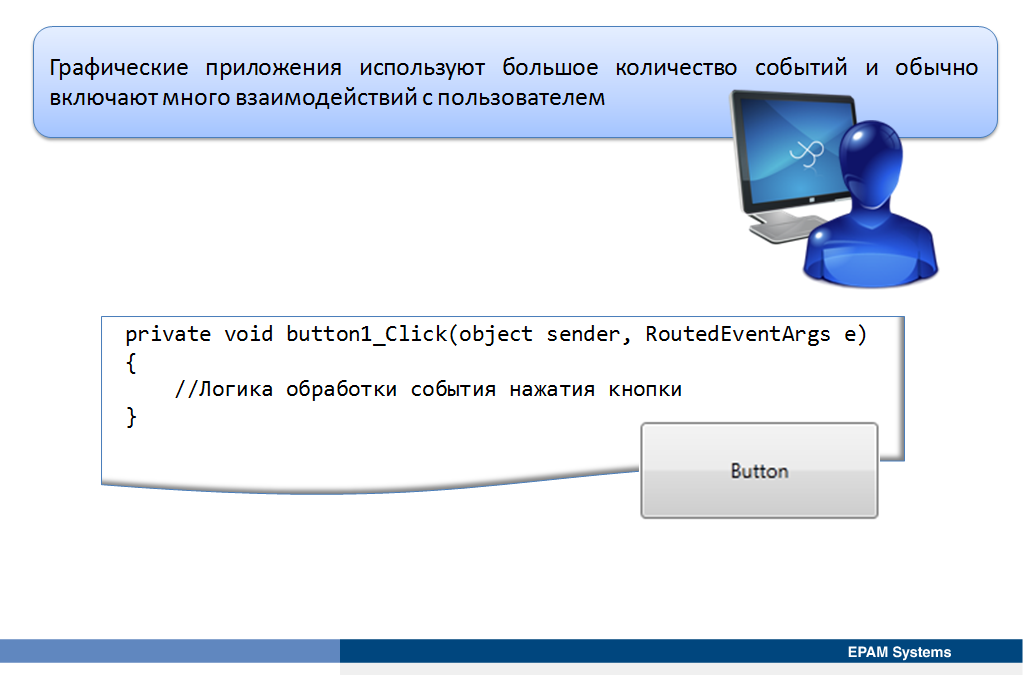
On метод должен быть защищенным и виртуальным (protected и virtual). Это позволяет любому из производных классов изменить процесс, когда возникает событие, например, при добавлении проверки свойства в аргументах события и генерации исключения, если значение не является действительным. При использовании защищенного виртуального метода генерируется исключение, если аргумент не проходит проверку достоверности; метод можно изменить в любом дочернем классе, если правила проверки меняются. Если используется On метод, новые проверки будут применяться ко вхождениям, где вызывается событие в типе и любым вхождениям, где событие возникает в дочернем классе.

**Практический совет: не следует отправлять нулевые значения событиям.** При генерации события следует избегать передачи null в качестве параметра и всегда отправлять экземпляры объектов. Для параметра sender нужно обычно использовать ключевое слово this. Не следует пренебрегать параметром sender, поскольку много приложений часто зависят от этого параметра.

## Демонстрация: Использование событий



## Использование событий в графических приложений



Графические приложения используют большое количество событий и обычно включают много взаимодействий с пользователем. Например, при нажатии кнопки пользователь ожидает незамедлительной реакции приложения. При разработке графических приложений используются события, немедленно реагирующие на взаимодействие пользователя с интерфейсом. Например, кнопка предоставляет событие Click, и, подписавшись на событие Click, можно добавить обработчик события, в котором поместить код для обработки события нажатия кнопки, выполняющий соответствующую логику.

При написании простого приложения, как правило, используется один поток, в котором последовательно выполняется весь код. Если процесс занимает длительное время, приложении будет приостановлено, ожидая завершения процесса. Часто разрабатываются приложения «зависание» которых в ожидании завершения длительно-работающего процесса не желательно. В .NET Framework эту проблему можно решить путем использования нескольких потоков. Тогда каждый поток может работать одновременно с другими потоками, что означает, что приложению больше не нуждается в ожидании завершения длительного процесса. Длительный процесс может быть выполнен в другом потоке, в то время как приложение может продолжать работу с другими задачами.

Графические приложения в .NET Framework используют потоки как и любые другие приложения. В .NET Framework все взаимодействие с пользовательским интерфейсом (UI) должно быть выполнено в одном потоке, часто упоминаемом как поток пользовательского интерфейса. Потоком пользовательского интерфейса является только поток, который может обновлять пользовательский интерфейс или реагировать на события, его порождающие.

Наряду с потоком пользовательского интерфейса, можно использовать другие потоки чтобы убедиться, что графическое приложение остается чувствительным; при этом, данные нужно размещать между потоками в определенном порядке для обеспечения их безопасности. Эту логику можно осуществить вручную или использовать поток BackgroundWorker.

Класс BackgroundWorker позволяет выполнять операции в отдельном, выделенном потоке. Операции, требующие много времени, такие как загрузка и транзакции базы данных, могут создавать впечатление, что пользовательский интерфейс перестал отвечать на действия пользователя. Если необходимо обеспечить быстрое реагирование пользовательского интерфейса, а подобные операции приводят к длительным задержкам, эффективным решением может стать класс BackgroundWorker.

Класс BackgroundWorker включает события, указывающие на прогресс в потоке. Пользовательский интерфейс может подписаться на эти события и обновлять экран с помощью потока пользовательского интерфейса. Это позволяет создавать пользовательский интерфейс, который остается гибким и актуальным.

Хотя класс BackgroundWorker хорошо работает с пользовательскими интерфейсами, его использование не ограничено этой возможностью. Объект BackgroundWorker можно использовать всякий раз, когда нужно запустить код в отдельном потоке и сообщать о прогрессе в этом коде.

**Для использования класса BackgroundWorker необходимо выполнить следующие действия:**

* Создать экземпляр класса BackgroundWorker.
* Добавить обработчик события DoWork. Обработчик события для события DoWork должен содержать код, который необходимо выполнить в отдельном потоке. При желании можно использовать метод ReportProgress объекта BackgroundWorker для сообщения о статусе операции.
* Можно добавить обработчик для события ProgressChanged. Обработчик события ProgressChanged работает в потоке пользовательского интерфейса, таким образом можно добавить код для обновления пользовательского интерфейса. Событие ProgressChanged возникает всякий раз, когда вызывается метод ReportProgress.
* Можно добавить обработчик для события RunWorkerCompleted. Это событие возникает, когда метод, связанный с событием DoWork, завершается. Обработчик для события RunWorkerCompleted выполняется в потоке пользовательского интерфейса, так что можно обновить пользовательский интерфейс.
* Если используется метод ProgressChanged, установить свойство WorkerReportsProgress в true.
* Вызвать метод RunWorkerAsync объекта BackgroundWorker. Этот метод вызывает событие DoWork и начинает выполнение фонового потока с помощью метода, указанного для этого события.

Класс BackgroundWorker обеспечивает метод CancelAsync. Этот метод задает логическое свойство CancellationPending и запрашивает метод, как выполняемый результат, когда событие DoWork завершается. Метод, который выполняет событие DoWork, периодически проверяет свойство CancellationPending, и когда оно устанавливается в true, метод завершается.

В следующем примере показано использование класса BackgroundWorker в обработчике события Click кнопки для запуска длительного процесса без «замораживания» пользовательского интерфейса. В примере также показано, как можно ответить на событие ProgressChanged, сохранив пользовательский интерфейс обновляемого приложения.

using System.ComponentModel;

using System.Windows;

namespace BackgroundWorkerClassViaWPF

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void button1\_Click(object sender, RoutedEventArgs e)

{

BackgroundWorker bw = new BackgroundWorker();

// Alternatively you can omit the parameter types.

bw.DoWork += ((object doWorkSender, DoWorkEventArgs doWorkArgs) =>

{

for (int i = 1; i <= 100; i++)

{

System.Threading.Thread.Sleep(50);

bw.ReportProgress(i \* 10);

}

});

bw.ProgressChanged += ((object progressChangedSender,

ProgressChangedEventArgs progressChangedArgs) =>

{

// Update label in UI with progress.

statusBox.Content = progressChangedArgs.ProgressPercentage.ToString() + "%";

});

bw.RunWorkerCompleted += ((object runWorkerCompletedSender,

RunWorkerCompletedEventArgs runWorkerCompletedArgs) =>

{

// Alternatively update the user interface.

MessageBox.Show("Complete");

});

bw.WorkerReportsProgress = true;

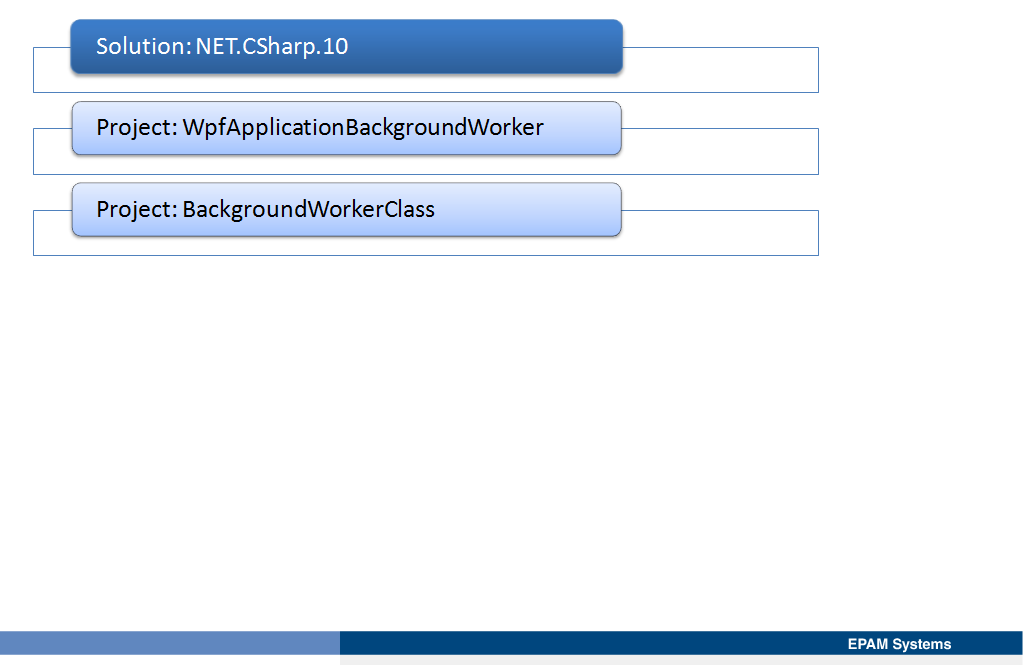
bw.RunWorkerAsync();

}

}

}

## Демонстрация: Использование событий в графических приложений



1. Наследовать напрямую от класса MulticastDelegate нельзя, для этого используется ключевое слово delegate. [↑](#footnote-ref-1)