|  |  |
| --- | --- |
| <<[Показать меню](http://www.rsdn.ru/?article/dotnet/delegat.xml) | **[http://www.rsdn.ru/images/printer2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml?print)**[**Сообщений 29**](http://www.rsdn.ru/forum/dotnet/876042.aspx)**[http://www.rsdn.ru/forum/images/showfr.gif](http://www.rsdn.ru/forum/dotnet/876042.aspx)**[**Оценка 1378**](http://www.rsdn.ru/Forum/RateList.aspx?mid=876042)**[http://www.rsdn.ru/images/fav.gif](http://www.rsdn.ru/Users/Private/AddFav.aspx?mid=876042)  [http://www.rsdn.ru/forum/images/sub.gif](http://www.rsdn.ru/Forum/Private/Subscr.aspx?tid=876042)   Оценить [http://www.rsdn.ru/forum/images/n11.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-3)[http://www.rsdn.ru/forum/images/n1.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=1)[http://www.rsdn.ru/forum/images/n2.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=2)[http://www.rsdn.ru/forum/images/n3.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=3)[http://www.rsdn.ru/forum/images/nx.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-1)[http://www.rsdn.ru/forum/images/np.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-4)[http://www.rsdn.ru/forum/images/nm.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=0)** |

[Система Orphus](http://orphus.ru/)

**Делегаты и события**

**Автор:**[***Алексей Дубовцев***](http://rsdn.ru/Users/Profile.aspx?uid=2592) ***The RSDN Group*  
Источник:**[***RSDN Magazine #4-2004***](http://www.rsdn.ru/mag/main.htm)

Опубликовано: *26.03.2005*  
Исправлено: *23.05.2006*  
Версия текста: *1.0*

|  |  |
| --- | --- |
| [**Введение в делегаты**](http://www.rsdn.ru/article/dotnet/delegat.xml#EJB)  [Общие сведения](http://www.rsdn.ru/article/dotnet/delegat.xml#EMB) [Виды методов](http://www.rsdn.ru/article/dotnet/delegat.xml#E3B)  [**Делегаты – начинаем непосредственную работу**](http://www.rsdn.ru/article/dotnet/delegat.xml#EFC)  [Создаем собственный делегат](http://www.rsdn.ru/article/dotnet/delegat.xml#EIC) [Делегат и экземплярные методы](http://www.rsdn.ru/article/dotnet/delegat.xml#EMG)  [**MulticastDelegate**](http://www.rsdn.ru/article/dotnet/delegat.xml#EYBAC)  [Пример использования свойств Method и Target](http://www.rsdn.ru/article/dotnet/delegat.xml#E1FAC) [Операторы сравнения](http://www.rsdn.ru/article/dotnet/delegat.xml#EVKAC) ["Нулевые делегаты"](http://www.rsdn.ru/article/dotnet/delegat.xml#EPTAE)  [**События**](http://www.rsdn.ru/article/dotnet/delegat.xml#EMUAE)  [Как устроены события и зачем они нужны](http://www.rsdn.ru/article/dotnet/delegat.xml#ERUAE) [События .NET](http://www.rsdn.ru/article/dotnet/delegat.xml#ERYAE) [Внутренний механизм поддержки событий](http://www.rsdn.ru/article/dotnet/delegat.xml#EG2AE) [Контроль над событиями](http://www.rsdn.ru/article/dotnet/delegat.xml#EQ5AE)  [**Дополнительные возможности при работе с делегатами**](http://www.rsdn.ru/article/dotnet/delegat.xml#EADAG)  [Список делегатов — EventHandlerList](http://www.rsdn.ru/article/dotnet/delegat.xml#EFDAG) [Стандартный делегат общей библиотеки](http://www.rsdn.ru/article/dotnet/delegat.xml#EZJAG)  [**В заключение**](http://www.rsdn.ru/article/dotnet/delegat.xml#EMLAG) | http://rsdn.ru/res/book/net/dotnetoriginal.pnghttp://www.rsdn.ru/mag/cover/mag0404.jpg |

Одна из глав книги ["Microsoft .NET в подлиннике"](http://rsdn.ru/article/?668)

Введение в делегаты

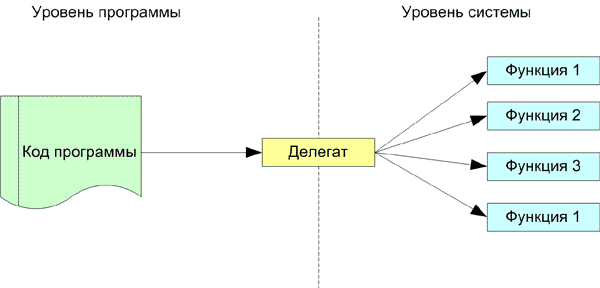
**Общие сведения**

Делегаты являются ссылками на методы, инкапсулирующими настоящие указатели и предоставляющими удобные сервисы для работы с ними. Ссылки представляют собой объекты соответствующего типа. Все делегаты являются объектами типа System.Delegate или System.MulticastDelegate, который является производным от первого.

Уже к окончанию разработки среды исполнения, ее архитекторы пришли к выводу, что целесообразно использовать только класс MulticastDelegate, а класс Delegate является избыточным. Но, опасаясь серьезных архитектурных нестыковок, разработчики были вынуждены оставить класс Delegate в составе общей библиотеки классов.

Различие между этими классами заключается в том, что экземпляры первого (делегаты) могут хранить лишь одну ссылку на метод, а экземпляры второго могут содержать сразу несколько ссылок на методы. Благодаря этому, можно подсоединять к одному делегату несколько методов, каждый из которых при единственном обращении к делегату будет вызваться по цепочке. Таким образом, из программы будет виден лишь один делегат, за которым скрывается несколько методов (рисунок 1).

Эта возможность очень удобна для поддержки событий, поскольку позволяет без использования дополнительных механизмов присоединить к событию несколько функций обработчиков. Фактически, делегат представляет собой объект — черный ящик, скрывающий в своих недрах указатели на функции. Важно понять, что делегаты, по сути дела, ничем не отличаются от обычных пользовательских объектов. Главная их особенность состоит лишь в том, что они имеют поддержку со стороны среды исполнения. Об их свойствах, в отличие от обычных объектов, знают даже компиляторы, предоставляющие удобные специальные сервисы для работы с ними.

  
*Рисунок 1. Делегат может ссылаться на несколько методов или функций.*

**Виды методов**

Все методы в среде .NET можно разделить на две группы: статические (static) и экземплярные (instance).

Если делегат ссылается на статический метод, то все действительно просто. Так как в этом случае есть вся необходимая для вызова метода информация: адрес метода и параметры. Если же делегат ссылается на экземплярный метод, то задача усложняется. Чтобы вызвать экземплярный метод, делегату необходимо знать ссылку на объект, к которому привязан данный конкретный метод. Оказывается, что эта ссылка хранится в самом объекте делегата и указывается при его создании. На протяжении всей жизни объекта делегата данная ссылка не изменяет своего значения, она всегда постоянна и может быть задана только при его создании. Таким образом, вне зависимости от того, ссылается ли делегат на статическую функцию или на экземплярный метод, обращение к нему извне ничем отличаться не будет. Всю необходимую функциональность обеспечивает сам делегат, вкупе со средой исполнения. Это очень удобно, поскольку множество разных делегатов можно привязывать к одному событию.

Делегаты – начинаем непосредственную работу

**Создаем собственный делегат**

Итак, делегат представляет собой экземпляр пользовательского класса, являющегося потомком класса MulticastDelegate. Соответственно, необходимо уметь объявлять подобные классы. Напрямую проделать операцию наследования от класса MultiCastDelegate или Delegate не получится, при попытке сделать это, компилятор выдаст следующую ошибку.

|  |
| --- |
| error CS0644: 'Hello' cannot inherit from special class 'System.Delegate' |

В переводе на русский язык сообщение означает: "Hello не может быть наследником специального класса System.Delegate". Наследование запрещает сам компилятор и только для этих классов, потому что в большинстве компиляторов предусмотрены специальные средства для работы с делегатами. В C# это специальная конструкция, начинающаяся с ключевого слова **delegate**.

При описании делегата мы должны указать прототип метода, на который будут ссылаться экземпляры данного делегата. В общем виде описание делегата будет выглядеть следующим образом.

|  |
| --- |
| delegate void MyDelegate(string s); |

Описывая делегат, необходимо понимать, что вводится новый тип, а именно, класс-потомок MulticastDelegate. То есть конструкция, приведенная выше, на самом деле гипотетически порождает примерно такой код:

|  |
| --- |
| class MyDelegate : MulticastDelegate  {  ...  // Здесь специальным образом закодирован прототип метода, на который  // мы собираемся ссылаться при помощи экземпляров делегата. |

Компилятор проводит такую трансляцию неявно, дабы не пугать рядовых программистов ужасами внутренней архитектуры делегатов. Таким образом, для того чтобы воспользоваться делегатом, необходимо создать экземпляр данного класса, который бы указывал на нужный нам метод. Сам по себе делегат является типом и никаких ссылок на метод содержать не может, а соответственно, не может и использоваться для обращения к ним. Для этого необходимо использовать экземпляр делегата.

Создание экземпляра делегата просто, поскольку компилятор уже подготовил все необходимое. Он создал специальный конструктор, которому нужно передать указатель на метод, а со всем остальным разберется среда исполнения.

Создание экземпляра делегата происходит следующим образом:

|  |
| --- |
| MyDelegate del = new MyDelegate(MyHandler); |

Где MyDelegate — это тип делегата, del — экземпляр делегата, который будет создан в результате выполнения конструкции, MyHandler — метод, на который будет ссылаться этот делегат. Соответственно, после создания экземпляра делегата можно обращаться к методам, на которые он ссылается. В языках высокого уровня существует возможность обращаться к экземпляру делегата, как к самому методу. Выглядеть это будет так.

|  |
| --- |
| del("Hello, world!"); |

Эта строка вызовет метод MyHandler, на который ссылается наш делегат. В завершение приведу пример, который будет аккумулировать сказанное ранее и наглядно демонстрировать работу с простейшим делегатом.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 2. Простейший пример работы с делегатами.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  // Основной класс приложения.  class App  {  // Опишем собственный делегат.  // В этом случае — это не что иное, как описание  // вложенного класса, просто оно замаскировано  // при помощи специального ключевого слова языка.  // Соответственно, для того чтобы использовать данный  // класс, придется создать его экземпляр.  delegate void MyDelegate(string s);  // Это функция, которую мы будем использовать для проверки  // работоспособности делегата.  // Она является статической и для ее вызова  // не требуется наличия экземпляра класса.  static void MyHandler(string s)  {  // Просто выведем на консоль единственный аргумент,  // переданный функции.  Console.WriteLine(s);  }  // Точка входа в приложение.  static void Main()  {  // Создадим экземпляр нашего делегата (типа! (класса!)),  // описанного нами ранее, передав ему в качестве  // параметра конструктора ссылку на функцию,  // которую мы хотим связать с делегатом.  MyDelegate del = new MyDelegate(MyHandler);  // Вызовем функцию через делегат,  // как видите, все достаточно просто, мы можем пользоваться  // экземпляром делегата как функцией.  del("Hello World");  }  }; |

В результате работы приложения на консоль будет выведена следующая строка:

|  |
| --- |
| Hello, World! |

Возникает естественный вопрос: "А к чему, собственно, такие сложности, не проще ли напрямую обратиться к методу MyHandler?" Оказывается, не проще. Хотя для данного примера, действительно, проще, поскольку мы точно знаем, какой метод нам нужно вызвать. Но для проектирования обратной связи с внешними системами делегаты незаменимы. Мы просто передаем внешней системе делегат, ссылающийся на наш внутренний метод. И покорно ждем, когда внешняя система обратится к нашей внутренней функции через переданный делегат.

**Делегат и экземплярные методы**

Теперь рассмотрим такой важный вопрос, как вызов экземплярных методов. Единственным отличием от обращения к статическим функциям будет способ создания делегата. Необходимо будет указать ссылку на объект, который будет использоваться при вызове метода через данный делегат.

|  |
| --- |
| MyDelegate del = new MyDelegate(sc.MyHandler); |

Где scявляется ссылкой на экземпляр объекта, который должен использоваться при обращении к методу MyHandler.

Приведу пример, демонстрирующий работу с подобными делегатами. Дабы излишне не запутывать читателя, а также упростить чтение листинга примера, в него введен дополнительный класс. В классе описан метод, на который будет ссылаться делегат. Данный метод будет использовать поле класса, что позволит нам следить за работой метода, изменяя его (листинг 3).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 3. Использование делегата для обращения к экземплярному методу.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class SomeClass  {  // Поле класса.  // Ни у кого не должно возникать сомнений, что оно  // напрямую связано с экземпляром данного типа.  public string SomeField;  // Обычный метод, связанный с экземпляром типа.  // Его будем вызывать при помощи делегата.  public void MyHandler(string s)  {  // Выведем на консоль значение поля SomeField, а также  // аргумент, переданный методу.  Console.WriteLine(SomeField + s);  }  };  // Основной класс приложения.  class App  {  delegate void MyDelegate(string s);  static void Main()  {  // Создадим экземпляр тестового класса.  SomeClass sc = new SomeClass();  // Создадим экземпляр делегата, содержащего,  // помимо ссылки на сам метод, также ссылку  // на объект, для которого будет вызван метод.  MyDelegate del = new MyDelegate(sc.MyHandler);  // Изменяем значение поля тестового объекта.  sc.SomeField = "Hello, World!";  // Вызовем метод через делегат.  del(" - from Instance Delegate");  // Эквивалентен следующему вызову  // sc.MyHandler(" - from Instance Delegate");  // Снова изменим значение поля тестового объекта.  sc.SomeField = "Good bye, World!";  // Снова обратимся к методу.  del(" - from Instance Delegate");  }  }; |

В результате работы приложения на консоль будут выведены следующие строки.

|  |
| --- |
| Hello, World! - from Instance Delegate  Good bye, World! - from Instance Delegate |

Видно, что метод действительно был вызван от нужного объекта. Изменения, внесенные в объект после первого обращения к методу, естественно, отразились на втором обращении к нему.

Теперь, после того как мы умеем пользоваться делегатами и знаем, для чего они нужны, приступим к их более детальному изучению.

MulticastDelegate

Этот класс является неотъемлемым в жизни любого делегата. Он предоставляет основные сервисы по управлению делегатами. Хотя многие из рассматриваемых здесь его членов все же описаны в базовом классе Delegate. Но в данном случае это можно отнести к конструкционным особенностям и не обращать на них внимания. Ниже, в таблице 1, приведено краткое описание наиболее важных членов класса MulticastDelegate.

**Свойства**

|  |  |  |
| --- | --- | --- |
| http://www.rsdn.ru/article/dotnet/delegat/image001.gif | Method | Возвращает метод, на который ссылается делегат |
| http://www.rsdn.ru/article/dotnet/delegat/image001.gif | Target | Возвращает объект, к которому привязан метод, на который ссылается делегат |

**Методы**

|  |  |  |
| --- | --- | --- |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gif | DynamicInvoke | Позволяет динамически обратиться к функциям, связанным с делегатом |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gif | GetInvocationList | Возвращает массив делегатов, привязанных к делегату, в порядке, в котором они вызываются. |
| http://www.rsdn.ru/article/dotnet/delegat/image003.gifhttp://www.rsdn.ru/article/dotnet/delegat/image004.gif | Equality Operator | Оператор (==), позволяет определить равенство делегатов |
| http://www.rsdn.ru/article/dotnet/delegat/image003.gifhttp://www.rsdn.ru/article/dotnet/delegat/image004.gif | Inequality Operator | Оператор (!=), позволяет определить, различны ли делегаты. |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gifhttp://www.rsdn.ru/article/dotnet/delegat/image004.gif | Combine | Конкатенирует два (или более) делегата, создавая новый делегат, список вызовов которого включает списки объединяемых делегатов. Исходные делегаты не модифицируются. |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gifhttp://www.rsdn.ru/article/dotnet/delegat/image004.gif | Remove | Удаляет список вызовов одного делегата из списка вызовов другого. При этом создается новый делегат, список вызовов которого представляет собой результат удаления. Исходные делегаты не модифицируются. |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gifhttp://www.rsdn.ru/article/dotnet/delegat/image004.gif | CreateDelegate | Позволяет динамически создать делегат |

*Таблица 1. Описание членов класса MulticastDelegate.*

Обратите внимание, что многие из методов описаны как статические. Это очень важный момент, который мы рассмотрим несколько позднее.

**MulticastDelegate.Method**

Возвращает описание метода, на который ссылается делегат.

|  |
| --- |
| public MethodInfo Method { get; } |

Особенность свойства состоит в том, что если делегат содержит не одну ссылку на функцию, а несколько, то свойство всегда будет возвращать последнюю добавленную ссылку.

**MulticastDelegate.Target**

Возвращает объект, с которым связан метод, на который ссылается делегат.

|  |
| --- |
| public object Target { get; } |

Если делегат ссылается на статический метод, свойство вернет пустую ссылку (null). Особенность свойства состоит в том, что если делегат содержит не одну, а несколько ссылок на методы, то свойство вернет объект, связанный с последним из добавленных в очередь методов.

**Пример использования свойств Method и Target**

Для наглядной демонстрации возможностей рассмотренных свойств, приведу пример, использующий их (листинг 4). В нем будет объявлен простой тестовый класс (SomeClass), содержащий единственный экземплярный метод (InstanceMethod). В программе будет создан объект данного класса. Далее будет создан делегат и к нему присоединен метод InstanceMethod. Затем на консоль будет выведена информация о делегате, полученная при помощи свойств Method и Target.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 4. Пример использования свойств Method и Target.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class SomeClass  {  // Объявим в нем экземплярный метод.  public void InstanceMethod()  {  // Сообщим пользователю о том, что метод вызван.  Console.WriteLine("InstanceMethod was called – Hello, World!");  }  };  // Основной класс приложения.  class App  {  // Опишем делегат.  // Обратите внимание, его прототип должен совпадать с прототипом  // метода, на который он ссылается.  delegate void MyDelegate();  // Точка входа в приложение.  public static void Main()  {  // Создадим экземпляр тестового класса.  SomeClass sc = new SomeClass();  // Создадим экземпляр делегата  MyDelegate del = new MyDelegate(sc.InstanceMethod);  // Выведем на консоль тип объект, к которому привязан метод,  // на который ссылается делегат.  Console.WriteLine("Target type = {0}",  del.Target.GetType().ToString());  // Выведем на консоль информацию по методу, на который ссылается  // делегат.  Console.WriteLine("Method = {0}",del.Method.ToString());  Console.WriteLine();  // Ну и для завершения картины вызовем сам делегат.  del();  }  } |

В результате работы приложения на консоль будут выведены следующие строки.

|  |
| --- |
| Target type = SomeClass  Method = Void InstanceMethod()  InstanceMethod was called – Hello, World! |

В случае, когда делегат содержит ссылку только на один метод, использование этих свойств крайне примитивно. Если же делегат содержит более одной ссылки, такой подход к получению информации годиться не будет. Необходимо будет воспользоваться другими способами. Оба свойства будут возвращать информацию только по последнему методу, добавленному в список вызова делегата.

При использовании свойства Target необходимо помнить, что статические методы не имеют привязки к объектам и значение этого свойства всегда будет равно нулевой ссылке.

**MulticastDelegate.DynamicInvoke**

Метод позволяет динамически обратиться к делегату.

|  |
| --- |
| public object Delegate.DynamicInvoke(  // Аргументы, которые следует передать  // при вызове функций, связанных с делегатом.  object[] args  ); |

Практическая польза метода весьма сомнительна, тем не менее, приведем пример работы с ним. Чтобы разнообразить пример, попытаемся вызвать делегат с неверным количеством параметров и посмотрим, что получится (листинг 5).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 5. Динамическое обращение к делегату.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class App  {  delegate void MyDelegate();  // А это подопытная функция-обработчик, которая будет связана  // с делегатом.  static void Handler()  {  Console.WriteLine("Handler method was called – Hello, World!");  }  public static void Main()  {  // Создадим экземпляр делегата.  SomeDelegate sd = new SomeDelegate(Handler);  // Сначала корректно обратимся к делегату.  sd.DynamicInvoke(new object[0]);  // Теперь вызовем делегат, с неверным количеством  // параметров.  sd.DynamicInvoke(new object[2]);  }  }; |

В результате работы приложения сначала на консоль будет выведена строка.

|  |
| --- |
| Handler method was called – Hello, World! |

Затем будет выведено сообщение о возникшем исключении TargetParameterCountException, со следующим объяснением

|  |
| --- |
| System.Reflection.TargetParameterCountException: Parameter count mismatch. |

То есть нам сообщают, что при вызове делегата было передано неверное количество аргументов. Таким образом, при работе с методом DynamicInvoke необходимо иметь в виду, что могут произойти неприятности подобного рода.

**Операторы сравнения**

**(Equality и Inequality)**

Операторы позволяют выяснить, ссылаются ли два делегата на одну и ту же функцию.

|  |
| --- |
| public static bool operator == (Delegate d1, Delegate d2);  public static bool operator != (Delegate d1, Delegate d2); |

При сравнении делегатов учитывается не только метод, на который ссылается делегат, но и ссылка на объект, с которым связан метод.

**MulticastDelegate.Combine и MulticastDelegate.Remove**

Методы предназначены для поддержки делегатов, ссылающихся одновременно на несколько функций. Первый метод позволяет объединить несколько делегатов в один, в списке вызовов которого будут находиться списки вызовов всех скомбинированных делегатов.

|  |
| --- |
| // Объединяет два делегата в один.  public static Delegate Delegate.Combine(  // Первый делегат.  Delegate,  // Второй делегат.  Delegate  );  // Объединяет произвольное количество делегатов в один.  public static Delegate Delegate.Combine(  // Массив делегатов, которые будут  // объединены в один.  Delegate[]  ); |

Обратите внимание на то, что в результате работы метода будет возвращен новый делегат, который в своем списке вызовов будет содержать функции всех указанных делегатов. При этом старые делегаты при выполнении данной операции абсолютно не пострадают, их списки вызова останутся нетронутыми.

Наряду с возможностью объединения нескольких делегатов в один, существует обратная операция — изъятия указанных методов из списка вызовов некоторого делегата.

|  |
| --- |
| // Позволяет изъять из делегата связь с определенными функциями,  // представленные также делегатами.  public static Delegate Delegate.Remove(  // Делегат-источник.  Delegate source,  // Функции, связанные с этим делегатом,  // будут изъяты из делегата-источника.  Delegate value  ); |

Рассмотренный метод, также как и метод Combine, не изменяет значение параметров, переданных ему. Он создает абсолютно новый делегат, в который помещает результаты своей работы.

Разработчики среды .NET решили, что простым программистам будет затруднительно использовать эти методы, в силу их нетривиальности. Поэтому в компилятор C# были введены специальные средства поддержки делегатов. Вы можете оперировать с делегатами при помощи обычных операций сложения и вычитания (+,-,+=,-=). При этом компилятор автоматически будет генерировать код, обращающийся к методам Combine и Remove.

Для наглядности приведем пример, демонстрирующий работу с делегатами, ссылающихся на несколько методов (листинг 6).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 6. Работа с делегатами, ссылающимися на несколько методов.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class App  {  delegate void MyDelegate(string s);  // Это функция, на которую мы будем ссылаться при помощи  // экземпляра делегата.  static void MyHandler(string s)  {  // Просто выведем на консоль аргумент, полученный нашей  // функцией.  Console.WriteLine(s);  }  static void Main()  {  // Создадим экземпляр делегата, описанного ранее.  // В качестве единственного параметра конструктора  // передадим ссылку на метод MyHandler.  MyDelegate del = new MyDelegate(MyHandler);  // Создадим новый делегат и скомбинируем его  // с ранее созданным делегатом.  del += new MyDelegate(MyHandler);  // То же самое можно сделать следующим образом.  del = del + new MyDelegate(MyHandler);  // А теперь уберем из списка вызова делегата  // одну из ссылок на метод MyHanlder.  del -= new MyDelegate(MyHanlder);  // Затем обратимся к конечному делегату, естественно,  // что при этом по цепочке будут вызваны все методы,  // связанные с ним.  del("Hello, world!");  }  }; |

В результате работы приложения на консоль будут выведены следующие строки.

|  |
| --- |
| Hello, World!  Hello, World! |

Обратите внимание, что функция была вызвана два раза, хотя мы добавляли ее в список вызова делегата три раза. Но если учесть, что мы один раз изъяли ее из списка вызова, то все становится понятно. Интересно, что в списке вызова делегата все методы одинаковы, но этот факт для сервисов, оперирующих с делегатами, не имеет никакого значения.

Компилятор преобразует арифметические операции над делегатами в обращения к методам Combine и Remove. Чтобы наглядно продемонстрировать это, приведу листинг функции Main на языке IL, который получился после транслирования исходного кода компилятором языка C# (листинг 7).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 7. Код функции Main на языке IL.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| .method private hidebysig static void Main() cil managed  {  .entrypoint  .maxstack 4  .locals init (class App/MyDelegate V\_0)  // Загружаем нулевой указатель — функция статическая,  // не привязанная к экземпляру объекта.  ldnull  // Загружаем указатель на функцию.  ldftn void App::MyHandler(string)  // Создаем экземпляр делегата.  newobj instance void App/MyDelegate::.ctor(object,  native int)  stloc.0  ldloc.0  // Создаем еще один точно такой же делегат с одним  // методом в списке вызова.  ldnull  ldftn void App::MyHandler(string)  newobj instance void App/MyDelegate::.ctor(object,  native int)  // Здесь как раз и происходит комбинирование делегатов,  // на уровне языка С# это выглядело как оператор сложения (+=).  call class [mscorlib]System.Delegate  [mscorlib]System.Delegate::Combine(  class [mscorlib]System.Delegate,  class [mscorlib]System.Delegate)  castclass App/MyDelegate  stloc.0  ldloc.0  // Создаем еще один точно такой же делегат с одним  // методом в списке вызова.  ldnull  ldftn void App::MyHandler(string)  newobj instance void App/MyDelegate::.ctor(object,  native int)  // Здесь как раз и происходит комбинирование делегатов,  // на уровне языка С# это выглядело как оператор сложения (+).  call class [mscorlib]System.Delegate  [mscorlib]System.Delegate::Combine(  class [mscorlib]System.Delegate,  class [mscorlib]System.Delegate)  castclass App/MyDelegate  stloc.0  ldloc.0  // Создаем еще один точно такой же делегат с одним  // методом в списке вызова.  ldnull  ldftn void App::MyHandler(string)  newobj instance void App/MyDelegate::.ctor(object,  native int)  // А здесь происходит изъятие метода из списка вызова делегата,  // на уровне языка С# это выглядело как оператор вычитания (-=).  call class [mscorlib]System.Delegate  [mscorlib]System.Delegate::Remove(  class [mscorlib]System.Delegate,  class [mscorlib]System.Delegate)  castclass App/MyDelegate  stloc.0  ldloc.0  ldstr "Hello, world!"  // А здесь происходит обращение к делегату.  // Более подробно этот момент рассмотрим несколько позднее,  // хотя, взглянув на следующую строку, вы можете обо всем  // догадаться самостоятельно.  callvirt instance void App/MyDelegate::Invoke(string)  ret  } |

Понимая, что чтение IL-листингов может быть для некоторых читателей несколько утомительно, приведем код примера на языке C#, напрямую использующего методы Combine и Remove (листинг 8).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 8. Пример, явно использующий методы Combine и Remove.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class App  {  delegate void MyDelegate(string s);  // Это функция, на которую мы будем ссылаться при помощи  // экземпляра делегата.  static void MyHandler(string s)  {  // Просто выведем на консоль аргумент, полученный нашей  // функцией.  Console.WriteLine(s);  }  // Точка входа в приложение.  static void Main()  {  // Создадим экземпляр делегата, описанного выше.  // В качестве единственного параметра конструктора  // передадим ссылку на метод MyHandler.  MyDelegate del = new MyDelegate(MyHandler);  // Создадим новый делегат и скомбинируем его  // с ранее созданным делегатом.  //del += new MyDelegate(MyHandler);  del = (MyDelegate)Delegate.Combine(del,new MyDelegate(MyHandler));  // То же самое можно сделать следующим образом.  // del = del + new MyDelegate(MyHandler);  del = (MyDelegate)Delegate.Combine(del,new MyDelegate(MyHandler));  // А теперь уберем из списка вызова делегата  // одну из ссылок на метод MyHanlder.  //del -= new MyDelegate(MyHanlder);  del = (MyDelegate)Delegate.Remove(del,new MyDelegate(MyHandler));  // А так мы вызовем функцию через делегат.  // Как видите, все просто, мы можем пользоваться  // экземпляром делегата как функцией.  del("Hello, world!");  }  }; |

Результат работы данного примера нисколько не будет отличаться от предыдущего.

|  |
| --- |
| Hello, World!  Hello, World! |

Таким образом, использование методов Combine, Remove и арифметических операций, предоставляемых компилятором, по сути дела, ничем не отличается. Разве что обращение ко вторым более удобно, и программы, написанные с их использованием, читаются гораздо легче.

**Delegate.Invoke или что там внутри?**

Напомним, как выглядит механизм вызова делегата через обращение к методу Invoke.

|  |
| --- |
| ldstr "Hello, world!"  callvirt instance void App/MyDelegate::Invoke(string) |

Программистам, использующим языки высокого уровня, это может показаться странным, поскольку на их уровне обращение к делегату происходит как к обычной функции.

|  |
| --- |
| del("Hello, World!") |

Дело в том, что метод Invoke недокументирован. Он не является членом классов Delegate и MulticastDelegate. Соответственно, можно предположить, что это специальный метод, генерируемый компилятором. Для проверки предположения обратимся к IL-коду. Действительно, там обнаружится метод Invoke, его код приведен ниже.

|  |
| --- |
| .method public hidebysig virtual instance void  Invoke(string s) runtime managed  {  } |

Его изучение показывает, что метод лишен какого-либо кода. Он попросту пустой и ничего не может делать. Так может показаться, если не принимать во внимание спецификатор runtime, используемый при определении метода. Данный спецификатор указывает на то, что метод будет реализован по ходу исполнения программы самой средой исполнения. Это означает, что код, обрабатывающий вызов метода, располагается в недрах самой среды исполнения. А она прекрасно осведомлена о внутреннем устройстве делегата и справится с вызовом всех его методов без дополнительной помощи со стороны программиста.

Помимо этого можно отметить, что конструктор каждого делегата описывается подобным образом. Для примера приведем код конструктора одного из делегатов.

|  |
| --- |
| .method public hidebysig specialname rtspecialname  instance void .ctor(object 'object',  native int 'method') runtime managed  {  } |

Код, реализующий делегат, также находится внутри среды исполнения. Оказывается, метод Invoke отвечает не только за вызов делегата, но также за хранение информации о прототипе методов, на которые может ссылаться делегат. Не правда ли, изящное решение? Прототип закодирован в самом методе Invoke, то есть его прототип полностью совпадает с прототипом делегата. Таким образом, не надо прибегать к использованию дополнительных средств, вроде MethodInfo, для хранения информации о прототипе. К тому же, это страхует от возможности случайной передачи неверного количества или типа параметров при классическом обращении к делегату.

**MulticastDelegate.GetInvocationList**

Возвращает список делегатов, находящихся в списке вызовов делегата.

|  |
| --- |
| public sealed override Delegate[] GetInvocationList(); |

Несмотря на кажущуюся абстрактность метода, он имеет очень важное практическое применение.

Существует два недостатка механизма вызова функций, связанных с делегатом. И оба они связаны со случаем, когда в списке вызова делегата присутствует более одной функции.

* Функции, на которые ссылаются делегаты, могут возвращать значения. Но мы их получить не сможем, поскольку штатная функция вызова Invoke возвращает значение, которая вернула последняя из опрошенных функций.
* При обращении к делегату, содержащему в списке вызова несколько функций, вовсе не гарантируется, что все функции из списка будут вызваны. Если одна из них сгенерирует исключение, то работа метода Invoke будет прервана и остальные функции вызваны не будут.

Особое волнение вызывает вторая проблема, поскольку важные сообщения попросту могут не дойти до адресата. Что неприемлемо при разработке надежных систем. Рассмотрим пример (листинг 9).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 9. Демонстрация ненадежности делегатов, ссылающихся на несколько функций](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class App  {  delegate void MyDelegate(string s);  // Первый метод, на который мы будем ссылаться при помощи  // делегата. Именно он будет вызывать исключение,  // не позволяющее обратиться ко второму методу.  static void MyHandler(string s)  {  Console.WriteLine("Handler invoked");  // А здесь произойдет исключение.  throw new Exception();  }  // Второй метод, на который мы будем ссылаться из делегата.  static void MyHandler2(string s)  {  // Если вдруг эта строка будет выведена, значит  // предположения о ненадежности делегатов неверны.  Console.WriteLine("Handler2 invoked");  }  // Точка входа в программу.  static void Main()  {  // Создадим экземпляр описанного выше делегата.  MyDelegate del = new MyDelegate(MyHandler);  // Добавим к делегату еще одну ссылку на нашу функцию.  del += new MyDelegate(MyHandler2);  // Введем дополнительный защищенный блок,  // чтобы отлавливать происходящие исключения.  try  {  // Вызовем функции, связанные с делегатом.  del("Hello, world!");  }  // Введем обработчик исключения, который предотвратит "падение"  // программы.  catch (Exception ex)  {  // Сообщим пользователю о том, что произошло исключение.  Console.WriteLine("Problem! Exception caught");  }  }  }; |

В результате работы программы на консоль будут выведены следующие строки.

|  |
| --- |
| Handler invoked  Problem! Exception caught |

Таким образом, исходя из информации, представленной в логе, нетрудно понять, что в список вызова делегата было добавлено два метода, но вызван был только первый из них. Надеюсь, причины такого поведения программы понятны.

Обойти проблему можно следующим образом. Для этого придется воспользоваться методом GetInvocationList, чтобы организовать обращение к делегатам полностью в ручном режиме. Преимущество подхода заключается в том, что можно полноценно контролировать обращение к каждому методу, на который ссылается делегат. Код примера вы обнаружите в листинге 10.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 10. Надежный способ работы с делегатами, ссылающимися на несколько методов.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  class App  {  delegate void MyDelegate(string param);  // Первый метод, на который мы будем ссылаться при помощи  // делегата. Именно он будет вызывать исключение,  // не позволяющее обратиться ко второму методу.  static void Handler1(string param)  {  // Выведем на консоль значение переданного параметра,  // а также уведомим пользователя о том, что данный  // метод был вызван.  Console.WriteLine("Handler1 invoked, param = {0}",param);  // Преднамеренно выбросим исключение.  throw new Exception();  }  // Второй метод, на который мы будем ссылаться из делегата.  static void Handler2(string param)  {  // Сообщим пользователю о том, что метод  // был вызван, а также выведем на консоль  // значение переданного параметра.  Console.WriteLine("Handler2 invoked, param = {0}",param);  }  // Точка входа в программу.  static void Main()  {  // Создадим экземпляр описанного делегата.  MyDelegate del = new MyDelegate(Handler1);  // Присоединим к нему еще одну функцию.  del += new MyDelegate(Handler2);  // Последовательно пройдем по каждому делегату, входящему  // в список вызова ранее созданного делегата.  foreach (MyDelegate d in del.GetInvocationList())  {  // Обернем вызов функции в защищенный  // блок, что позволит предотвратить  // преждевременное завершение алгоритма.  try  {  d("Hello");  }  // Это блок обработки исключений, произошедших  // в защищенном блоке.  catch(Exception ex)  {  // Сообщим пользователю о том, что при попытке  // вызова одной из функций произошло исключение.  Console.WriteLine("Oh mama, some exception has occurred");  }  }  }  }; |

В результате работы программы на консоль будут выведены следующие строки.

|  |
| --- |
| Handler1 invoked, param = Hello  Oh mama, some exception has occurred  Handler2 invoked, param = Hello |

Видно, что задачи, поставленные перед приложением, были успешно выполнены. Несмотря на то, что в одной из функций, на которую ссылался делегат, было выброшено исключение, это не отразилось на работе остальных функций.

Помимо кода, обрабатывающего исключения, можно добавить специальные сервисы, анализирующие значения, возвращаемые функциями, вызываемыми через делегат.

Отметим, что обращаться к методам можно не только через функцию DynamicInvoke. Для этих целей также можно использовать свойство Method. А точнее, метод Invoke, предоставляемый классом MethodInfo, экземпляр которого можно получить через вышеупомянутое свойство класса Delegate. В итоге, код вызова можно заменить следующим:

|  |
| --- |
| // Было.  d.DynamicInvoke(param)  // Стало.  d.Method.Invoke(d.Target,param) |

Особую практическую пользу в этом подходе найти, конечно, сложно, особенно учитывая его медлительность. Но зато вы будете всегда иметь запасной вариант при работе с делегатами.

**MulticastDelegate.CreateDelegate**

Метод позволяет динамически создать делегат заданного типа.

|  |
| --- |
| public static Delegate Delegate.CreateDelegate(  Type type,  MethodInfo method);  public static Delegate Delegate.CreateDelegate(  Type type,  Object target,  string method);  public static Delegate Delegate.CreateDelegate(  Type type,  Type target,  String method);  public static Delegate Delegate.CreateDelegate(  Type type,  Object target,  String method,  bool ignoreCase); |

Этот метод введен в рамках поддержки технологии отражения, он позволяет динамически создавать делегаты произвольного типа. Он используется, если на момент компиляции не известно, сколько аргументов принимает метод обратного вызова и какого они типа.

**"Нулевые делегаты"**

При программировании делегатов может создаться впечатление, что существуют так называемые нулевые делегаты. Под нулевыми делегатами подразумеваются экземпляры классов делегатов, не содержащие ссылок на методы. Такой делегат можно попытаться получить, изъяв из списка все методы, на которые он ссылается при помощи функции Remove. В коде это будет выглядеть так.

|  |
| --- |
| del = new MyDelegate(SomeMethod);  del -= new MyDelegate(SomeMethod); |

И если до исполнения инструкции делегат ссылался на один метод SomeMethod, то в результате ее исполнения мы должны получить пустой делегат. На самом деле, мы получаем нулевую ссылку, то есть просто null. Что, кстати говоря, неочевидно, поскольку мы и далее имеем переменную del, которая подразумевает под собой ссылку на экземпляр делегата.

Большинство программистов попросту не ожидают такого подвоха со стороны системы и попадаются в коварную ловушку — они пытаются использовать методы данного экземпляра делегата. А поскольку он равен null, то происходит исключение NullReferenceException. Для того чтобы избежать подобных ошибок, в неоднозначных местах программы необходимо обязательно проверять значение делегата на null. Делается это следующим образом.

|  |
| --- |
| if (del != null)  del("Hello, World!\n"); |

Во всех случаях, когда значение делегата не известно со стопроцентной уверенностью, такая проверка обязательно должна присутствовать. Это позволит вам избежать очень неприятных ошибок.

События

В разделе будут рассмотрены члены типов — события.

**Как устроены события и зачем они нужны**

Начнем изучение с несколько идеализированного примера. Допустим, имеется некоторый компонент, и пускай для упрощения примера это будет банальная кнопка. Всем известно, что на кнопки можно нажимать. При нажатии кнопки происходит событие "щелчок", о котором необходимо обязательно уведомлять пользователя нашего компонента. Для этого введем в класс, представляющий компонент-кнопку, общедоступное поле Click, являющееся экземпляром делегата. И каждый раз, когда будет происходить соответствующее событие, будем обращаться к этому делегату. А он, в свою очередь, будет вызывать прикрепленные к нему функции и методы. Приведу код (листинг 11).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 11. Пример события на основе делегата.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| // Введем специальный делегат.  delegate void ClickHandler();  class Button  {  // Это общедоступное поле-делегат, к которому каждый  // может присоединить собственный метод.  public ClickHandler Click;  // Несколько идеализированная функция обработки  // сообщений, приходящих на кнопку.  void OnMsg(...)  {  // Предположим  switch(msg)  {  // Вот мы как бы засекли нажатие на кнопку.  case WM\_LBUTTONDOWN:  // Вызовем функции, связанные с нашим делегатом,  // предварительно проверив, а зарегистрирована  // ли хотя бы одна функция в поле-делегате.  if (Click != null)  Click();  }  }; |

Теперь пользователи нашего класса смогут присоединить свои функции к переменной-полю экземпляра делегата и получить уведомление о произошедшем событии.

Но поскольку среда .NET является объектно-ориентированной, мы обязаны соблюдать правила инкапсуляции полей. Соответственно, необходимо ввести дополнительные методы, обслуживающие поле-экземпляр делегата и контролирующие все производимые над ним операции, а само поле полагается сделать закрытым. Сделаем это следующим образом — добавим две функции add\_Click и remove\_Click, которые, соответственно, будут добавлять и убирать события в очередь делегата. Новый код представлен в листинге 12.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 12. Пример события на основе делегата, с поддержкой инкапсуляции полей.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| // Введем специальный делегат.  delegate void ClickHandler();  class Button  {  // Поле-делегат, к которому будут присоединяться обработчики  // нажатия кнопки.  // Обратите внимание, это поле теперь является закрытым,  // оно недоступно извне класса.  private ClickHandler Click;  // Введем два общедоступных метода,  // которые будут предоставлять сервисы работы с событием.  public add\_Click(ClickHandler delegate)  {  // Воспользуемся удобным арифметическим оператором  // для комбинирования делегатов.  Click += delegate;  }  // Удаление функции из списка вызова.  public remove\_Click(ClickHandler delegate)  {  // Удалим функции, представленные делегатом.  // delegate из нашего списка вызовов  Click -= delegate;  }  // Несколько идеализированная функция обработки  // сообщений, приходящих на кнопку.  void OnMsg(...)  {  // Предположим.  switch(msg)  {  // Вот мы как бы засекли нажатие кнопки.  case WM\_LBUTTONDOWN:  // Вызовем функции, связанные с нашим делегатом,  // предварительно проверив, а зарегистрирована  // ли хотя бы одна функция в поле-делегате.  if (Click != null)  Click();  }  }; |

Теперь наш компонент удовлетворяет основным парадигмам объектно-ориентированного программирования. Только вот, его код стал уж больно громоздким и его использование не будет столь наглядным, как прежде.

**События .NET**

Понимали это и разработчики, создававшие среду .NET. Для решения этой задачи они ввели специальное поле-событие, которое автоматически вводит поле-делегат и два дополнительных метода, обслуживающих его. Таким образом, автоматически реализуется представленная выше модель.

В языке высокого уровня C# определение таких полей осуществляется при помощи ключевого слова event. Также компилятор берет на себя заботу о работе с этими полями-событиями, благодаря чему к ним можно прибавлять и вычитать делегаты, хотя они, по сути дела, таковыми не являются. Приведем наглядный пример использования полей-событий (листинг 13).

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 13. Использование полей-событий.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  // Введем собственный делегат, не принимающий  // никаких значений.  delegate void MyDelegate();  // Это тестовый класс, он представляет собой гипотетический  // компонент кнопки.  class Button  {  // Введем общедоступное событие, к которому  // смогут подключаться все желающие.  public event MyDelegate Click;  // Данная функция необходима для того, чтобы  // симулировать событие нажатия на кнопку.  public void SimulateClick()  {  // Вызываем функции, связанные с событием Click,  // предварительно проверив, зарегистрировался  // ли кто-нибудь в данном событии.  if (Click != null)  Click();  }  };  class App  {  static void Main()  {  // Создаем экземпляр класса/компонента.  Button sc = new Button();  // Добавляем обработчик к его событию.  sc.Click += new MyDelegate(Handler);  // Сами вызовем функцию, которая инициирует  // возникновение события нажатия на кнопку.  sc.SimulateClick();  }  // А это функция-обработчик события нажатия на кнопку.  static void Handler()  {  Console.WriteLine("Hello, World!");  }  }; |

В результате работы приложения, на консоль будет выведена следующая строка:

|  |
| --- |
| Hello, World! |

Таким образом, работа с событиями представляется предельно простой. Она, по сути дела, ничем не отличается от взаимодействия с обыкновенными делегатами.

**Внутренний механизм поддержки событий**

Рассмотрим, какой код поддержки события был создан компилятором. Для этого изучим IL- код.

Во-первых, компилятор создал поле-делегат, в котором хранятся все зарегистрированные обработчики события.

|  |
| --- |
| .field private class MyDelegate Click |

Обратите внимание, что поле является закрытым и может быть использовано только из самого класса, исключая даже его потомков (private). Также компилятор создал одноименное специализированное свойство, в котором указаны методы, реализующие внешнюю работу с данным событием. Причем необходимо отметить, что это свойство по умолчанию является общедоступным.

|  |
| --- |
| .event MyDelegate Click  {  .addon instance void Button::add\_Click(class MyDelegate)  .removeon instance void Button::remove\_Click(class MyDelegate)  } |

Дабы не утомлять читателя излишним чтением IL-листингов, приведем код методов на языке C#.

|  |
| --- |
| public void add\_Click(MyDelegate del)  {  Click += del;  }  public void remove\_Click(MyDelegate del)  {  Click -= del;  } |

Методы будут использоваться только при работе с событием извне класса. Внутри же используется прямое обращение к полю Click, что несколько быстрее, чем вызов методов add\_Click и remove\_Click. Здесь можно усмотреть борьбу команды разработчиков компилятора С# за производительность создаваемых им программ.

Чтобы подтвердить сказанное, приведем IL-код функции SimulateClick, которая непосредственно обращается к событию из самого класса, если быть до конца точным, то не к самому событию, а к полю-экземпляру делегата.

|  |
| --- |
| .method public hidebysig instance void  SimulateClick() cil managed  {  .maxstack 1  ldarg.0  // Загружаем в стек указатель на интересующее нас поле.  ldfld class MyDelegate Button::Click  brfalse.s IL\_0013  ldarg.0  // Загружаем в стек указатель на интересующее нас поле.  ldfld class MyDelegate Button::Click  callvirt instance void MyDelegate::Invoke()  ret  } |

Обратите внимание на выделенную в коде инструкцию ldfld — она предназначена для работы с полями объекта. Также в листинге отсутствуют обращения к функциям add\_Click и remove\_Click. Что и требовалось доказать!

**Контроль над событиями**

При разборе первого примера подраздела говорилось, что введение дополнительных функций позволяет контролировать работу с событием. Но как оказалось, события .NET полностью закрыты для программиста. Они скрывают всю грязную работу за кулисами и контролировать обращение к событиям не представляется возможным. Но это утверждение верно только при стандартном способе использования событий. Также существует дополнительный расширенный режим их использования, при котором программист может самостоятельно объявить функции, управляющие подключением и отключением делегатов-обработчиков. Для этого в языке C# предусмотрена специальная конструкция, с использованием двух дополнительных ключевых слов add и remove.

|  |
| --- |
| event DelegateName SomeEvent  {  add  {  // Код, реализующий добавление делегата к списку  // вызова события.  }  remove  {  // Код, реализующий изъятие делегата из списка вызова  // события.  }  } |

Единственное, что может смутить при рассмотрении кода, так это то, что add и remove, по сути дела, являются функциями, принимающими один параметр. Чтобы получить доступ к этому параметру, необходимо воспользоваться ключевым словом value.

При использовании расширенного режима компилятор не будет автоматически описывать поле-делегат, и его придется вводить самостоятельно. Рассмотрим пример, демонстрирующий работу данного подхода (листинг 14). Этот пример по функциональности и внутренней архитектуре будет аналогичен предыдущему, только код поддержки работы с событием будет реализован в нем вручную.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 14. Поддержка событий в ручном режиме.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  // Опишем делегат, имеющий пустой прототип.  delegate void MyDelegate();  // Опишем подопытный класс/компонент.  class Button  {  // Закрытое поле-ссылка на экземпляр делегата,  // который будет обслуживать событие.  private MyDelegate \_click;  // Пользовательское событие, с возможностью контроля доступа  // к нему.  public event MyDelegate Click  {  // Эта функция будет вызвана при попытке добавления  // делегата в список вызова события.  add  {  // Сообщим пользователю о том, что произведена  // попытка добавить делегат.  Console.WriteLine("add handler was invoked");  // Добавим делегат в список вызова.  \_click += value;  }  // Эта функция будет вызвана при попытке изъятия  // делегата из списка вызова события.  remove  {  // Сообщим о том, что произведена попытка  // изъять делегат из списка вызова.  Console.WriteLine("remove handler was invoked");  // Удалим эту функцию из списка обработки.  \_click -= value;  }  }  // Эта функция необходима для того, чтобы  // симулировать событие нажатия на кнопку.  public void SimulateClick()  {  // Вызываем функции, связанные с событием Click,  // предварительно проверив, зарегистрировался  // ли кто-нибудь в этом событии.  if (Click != null)  Click();  }  };  class App  {  public static void Main()  {  // Создадим тестовый экземпляр компонента/класса.  Button sc = new Button();  // Добавим обработчик события.  sc.Click += new MyDelegate(Handler);  // Косвенно вызовем функцию нашего обработчика.  sc.SimulateClick();  // Уберем функцию-обработчик из списка вызова  sc.Click -= new MyDelegate(Handler);  // Попытаемся снова осуществить вызов.  sc.SimulateClick();  }  // Функция-обработчик для компонента/класса.  public static void Handler()  {  Console.WriteLine("Hello World - Handler was invoked");  }  }; |

В результате работы примера получим на консоли следующие строки.

|  |
| --- |
| add handler was invoked  Hello World - Handler was invoked  remove handler was invoked |

Как видим, приложение действовало по заданной схеме, контролируя при этом добавление и изъятие делегатов.

Такая возможность может быть весьма полезна для анализа и слежения за функциями, прикрепленными к событию. К примеру, можно запретить добавление событий кодом, не имеющим определенных привилегий защиты.

Дополнительные возможности при работе с делегатами

Здесь будут рассмотрены дополнительные возможности, предоставляемые общей библиотекой классов (FCL, Framework Class Library) для упрощения работы с делегатами. Некоторые из них могут быть весьма полезны в повседневном программировании.

**Список делегатов — EventHandlerList**

В рамках компонентной модели общей библиотеки классов введен дополнительный класс EventHandlerList. Он предназначен для упрощения разработки компонентов, содержащих большое количество событий. Класс позволяет хранить в своем экземпляре неограниченное количество событий, организуя доступ к ним при помощи произвольных ключей типа Object. Наиболее интересные члены класса EventHandlerList описаны в таблице 2.

|  |  |  |
| --- | --- | --- |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gif | AddHandler | Добавляет делегат в список по ключу |
| http://www.rsdn.ru/article/dotnet/delegat/image002.gif | RemoveHanlder | Изымает делегат из списка по ключу |

*Таблица 2. Члены класса System.ComponentModel.EventHandlerList.*

По сути дела, список является и не списком вовсе, а ассоциативным массивом. Но основное его достоинство заключается не в этом. Главное его отличие от обычных коллекций состоит в том, что он производит добавление и изъятие элементов из списка, учитывая особенности делегатов. Операции добавления и изъятия производятся при помощи методов Combine и Remove. Таким образом, по одному ключу может храниться несколько скомбинированных однотипных делегатов. Соответственно, при изъятии делегата по заданному ключу, элемент списка будет удален не полностью, а произойдет рекомбинация делегата, в ходе которой некоторые из его ссылок будут потеряны. Но сам делегат может не истощиться, а содержать еще несколько ссылок. Такое поведение списка весьма удобно при разработке компонентов, предоставляющих пользователям множество событий. Можно попросту добавлять и удалять делегаты по заданным ключам, не задумываясь о дополнительной поддержке событий.

Продемонстрируем работу со списком на примере того же компонента кнопки (листинг 15). На этот раз он будет содержать два события, делегаты которых будут храниться в специализированном закрытом списке.

[[http://www.rsdn.ru/images/ls2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml)Листинг 15. Использование специализированной коллекции EventHandlerList при работе с делегатами.](http://www.rsdn.ru/article/dotnet/delegat.xml)

|  |
| --- |
| using System;  // Подключим пространство имен, отвечающее за поддержку компонентной  // модели.  using System.ComponentModel;  // Опишем делегат с пустым прототипом.  delegate void MyDelegate();  // Введем тестовый класс/компонент.  class Button  {  // В этом закрытом списке будут храниться делегаты,  // представляющие наши события.  private EventHandlerList \_eventHandlers;  // Далее описаны два ключа, которые будут использоваться  // для индексации элементов внутри списка.  // Эти ключи определены с использованием идентификатора  // readonly, поскольку необходима стопроцентная гарантия,  // что ключи не изменят своего значения во время работы  // программы, иначе работа программы будет непредсказуема.  private static readonly object m\_MouseUpKey = new Object();  private static readonly object m\_MouseDownKey = new Object();  // Конструктор по умолчанию для класса.  public Button()  {  // Инициализируем список.  \_eventHandlers = new EventHandlerList();  }  // Опишем первое событие с поддержкой контроля.  public event MyDelegate MouseUp  {  add  {  // Добавим делегат в список по ключу.  \_eventHandlers.AddHandler(m\_MouseUpKey,value);  }  remove  {  // Удалим делегат из списка по ключу.  \_eventHandlers.RemoveHandler(m\_MouseUpKey,value);  }  }  // Опишем второе событие с поддержкой контроля.  public event MyDelegate MouseDown  {  add  {  // Добавим делегат в список по ключу.  \_eventHandlers.AddHandler(m\_MouseDownKey,value);  }  remove  {  // Удалим делегат из списка по ключу.  \_eventHandlers.RemoveHandler(m\_MouseDownKey,value);  }  }  // Данная функция будет вызывать первое событие.  public void SimulateMouseUp()  {  // Получаем нужный делегат из нашего списка  // при помощи ключа.  MyDelegate eh = (MyDelegate)\_eventHandlers[m\_MouseUpKey];  // Проверяем, присоединены ли функции к делегату, полученному из  // коллекции. В случае, если это так, вызываем делегат.  if (eh != null)  eh();  }  // А эта функция будет вызывать второе событие.  public void SimulateMouseDown()  {  // Получаем нужный делегат из нашего списка  // при помощи ключа.  MyDelegate eh = (MyDelegate)\_eventHandlers[m\_MouseDownKey];  // Проверяем, присоединены ли функции к полученному  // делегату. В случае, если это так, вызываем делегат.  if (eh != null)  eh();  }  };  class App  {  public static void Main()  {  // Создаем экземпляр тестового класса.  Button sc = new Button();  // Присоединяем обработчики к событиям.  sc.MouseUp += new MyDelegate(HandlerMouseUp);  sc.MouseDown += new MyDelegate(HandlerMouseDown);  // Вызываем события  sc.SimulateMouseUp();  sc.SimulateMouseDown();  }  // Обработчик для первого события.  public static void HandlerMouseUp()  {  // Уведомляем пользователя о произошедшем  // событии выводом соответствующего сообщения  // на консоль.  Console.WriteLine("HandlerUp was called");  }  // Обработчик для второго события.  public static void HandlerMouseDown()  {  // Уведомляем пользователя о произошедшем  // событии выводом соответствующего сообщения  // на консоль.  Console.WriteLine("HandlerDown was called");  }  }; |

В результате работы приложения на консоль будут выведены следующие строки.

|  |
| --- |
| HandlerUp was called  HandlerDown was called |

Как видно, нововведение никоим образом не отразилось на использовании класса извне. Была всего лишь изменена внутренняя организация компонента, несколько упростив работу программистам при помощи специализированного хранилища делегатов.

**Стандартный делегат общей библиотеки**

На протяжении главы всегда использовался собственноручно определенный делегат следующего вида:

|  |
| --- |
| delegate void MyDelegate(); |

Многие классы стандартной библиотеки используют делегаты для уведомления о произошедших в них событиях. Соответственно, разработчики среды .NET сочли, что будет целесообразно ввести общий стандартный тип делегата, который будет использоваться во всей библиотеке. Его прототип представлен ниже.

|  |
| --- |
| public delegate void EventHandler(  // Ссылка на объект, вызвавший событие.  object sender,  // Параметры, описывающие событие.  EventArgs e  ); |

Этот делегат принимает всего лишь два параметра, что явно маловато для универсального делегата. Но все же оказывается, параметров с лихвой хватает для передачи любой информации. Сам по себе класс EventArgs не содержит ни одного интересного члена, способного передавать какую бы то ни было информацию. Он введен лишь для обобщения. Когда необходимо передать дополнительную информацию, вводится новый класс, производный от EventArgs, в котором уже и вводятся поля, передающие необходимую информацию. В одной общей библиотеке у класса EventArgs 100 потомков, и их количество от версии к версии среды исполнения неуклонно растет.

При обращении к событию, имеющему тип данного делегата, в качестве параметра необходимо передать ссылку на текущий объект (this), а во втором параметре экземпляр класса EventArgs или производного от него.

|  |
| --- |
| SomeEvent(this, EventArgs.Empty); |

где свойство Empty попросту возвращает пустой экземпляр типа EventArgs. В принципе, его можно создать и самому воспользовавшись оператором new.

|  |
| --- |
| SomeEvent(this, new EventArgs()); |

Правда, такой подход будет работать несколько медленнее, чем первый, поскольку здесь появилась дополнительная операция создания объекта. И причем необходимо отметить, что она далеко из не самых быстродействующих.

В заключение

Использование делегатов и событий в обычных управляемых приложениях несложно. Выше были рассмотрены все наиболее важные аспекты проблемы. Но при проектировании производительных и устойчивых приложений, можно задуматься о создании собственного механизма обращения к делегатам, с использованием многопоточной модели и более защищенных алгоритмов.

*Эта статья опубликована в журнале RSDN Magazine #4-2004. Информацию о журнале можно найти*[*здесь*](http://www.rsdn.ru/mag/main.htm)

|  |  |
| --- | --- |
| <<[Показать меню](http://www.rsdn.ru/?article/dotnet/delegat.xml) | **[http://www.rsdn.ru/images/printer2.gif](http://www.rsdn.ru/article/dotnet/delegat.xml?print)**[**Сообщений 29**](http://www.rsdn.ru/forum/dotnet/876042.aspx)**[http://www.rsdn.ru/forum/images/showfr.gif](http://www.rsdn.ru/forum/dotnet/876042.aspx)**[**Оценка 1378**](http://www.rsdn.ru/Forum/RateList.aspx?mid=876042)**[http://www.rsdn.ru/images/fav.gif](http://www.rsdn.ru/Users/Private/AddFav.aspx?mid=876042)  [http://www.rsdn.ru/forum/images/sub.gif](http://www.rsdn.ru/Forum/Private/Subscr.aspx?tid=876042)   Оценить [http://www.rsdn.ru/forum/images/n11.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-3)[http://www.rsdn.ru/forum/images/n1.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=1)[http://www.rsdn.ru/forum/images/n2.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=2)[http://www.rsdn.ru/forum/images/n3.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=3)[http://www.rsdn.ru/forum/images/nx.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-1)[http://www.rsdn.ru/forum/images/np.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=-4)[http://www.rsdn.ru/forum/images/nm.gif](http://www.rsdn.ru/forum/Private/Rate.aspx?mid=876042&rate=0)** |