Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

Объектно-ориентированный анализ и программирование на языке **C# (C_Sharp)**

Материалы к 7-й лекции. Дополнение **2**

Проф. Забудский Е.И.

Москва 2007

Лекция 7. Дополнение *2*

Тема 9	Программная модель **Windows Forms** – основа для разработки приложений **.NET Framework** с графическим интерфейсом пользователя
	В дополнении **2** к Лекции 7 рассмотрены делегаты (delegate) и события (event)

см. также Материалы к Практ. зан. 8

NB	**Задание #2 на дом по итогам 2-го модуля (продолжение задания №1):** **Написать C#-программу – Банковский счет – применение полиморфизма.** **Программу реализовать в среде MS VS .NET 2005. Получить результат в консольном варианте.** **См. с. 38…40 в Материалах к 7-й лекции**
	Срок представления кода и результата – 21 февраля 2007 г.

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика.

Необходимым условием усвоения дисциплины является ВАША самостоятельная работа

Советую Вам все материалы, подготовленные мной к лекциям и практическим занятиям, распечатать и прорабатывать их! Приведенные C#-программы реализовать в среде MS VS .NET 2005 и разобраться в них.

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены **C#** и платформа **.NET** (step by step).

Делегаты и события (Дополнение 2 к лекции 7)

Содержание

NB	Рекомендую перед рассмотрением материала, изложенного в разделах 10 и 11 (Лекция 7, с. 25…33), изучить Дополнения 1 и 2

Делегаты и события (Дополнение 2 к лекции 7)

Введение

Посредством механизма динамического связывания осуществляется автоматический вызов реализации метода для объекта, на который указывает переменная в момент вызова. То есть имеется возможность прямо во время исполнения задействовать различные реализации одним и тем же вызовом метода. Это практично, поскольку при программировании заранее не известно, на объекты какого подкласса во время исполнения будет указывать переменная типа-предка. Теперь решение этой задачи можно будет отложить до момента исполнения программы.

Делегат, как наследование, интерфейс и полиморфизм, также **помогает отложить решение, о реализации методов, до времени исполнения**.

Применение делегата похоже на применение абстрактного метода. **Он, по аналогии, определяет тип возвращаемого значения и его формальные параметры**, **но не определяет саму реализацию**. Один и тот же делегат может представлять различные пути реализации метода во время исполнения. Но это возможно только тогда, когда эти методы имеют те же формальные параметры, и тот же тип возвращаемого значения, что и указанные в делегате. **Во время исполнения делегату присваивается подходящий метод**. **Метод затем инкапсулируется**. **Теперь, когда происходит вызов делегата, управление передается** (**делегируется**) непосредственно инкапсулированному методу.

Применение делегата весьма полезно и тогда, когда во время написания исходного кода **известно, что действие** (вместе с типом возвращаемого им значения и формальными параметрами) **должно произойти в определенной точке кода**, **но сам путь реализации его неизвестен**. Эта проблема часто встречается при написании современных компьютерных программ. Один из наглядных примеров — управляемый событиями **Г**рафический **П**ользовательский **И**нтерфейс (**GUI**). Каждое последующее действие в программе, управляемой событиями, зависит от вида возникшего события. Событием может быть щелчок кнопки мыши или нажатие клавиши на клавиатуре. **Бывает, точно известно, что кнопка при нажатии должна выполнить некоторое действие**, **но на момент компиляции еще не известно, какое именно**. Если же по нажатию кнопки предусмотреть вызов делегата, момент принятия решения о вызванном методе, будет **перенесен на стадию исполнения** программы. **Делегату во время выполнения можно присвоить конкретный метод, который будет в дальнейшем вызываться при каждом нажатии на кнопку**.События и делегаты тесно связаны между собой. Оба этих понятия являются основополагающими для написания программ, управляемых событиями, на языке C#.

1. Делегаты

Делегат является классом, вследствие этого, ссылочным типом и происходит от базового класса System. Delegate. Как и любой другой класс, делегат должен быть определен и впоследствии может быть порожден.

Примечание

В номенклатуре ООП, термин "класс" служит для определения класса, содержащегося в исходном тексте. Его **порождение**, происходящее во время исполнения программы, **называется объектом**. К сожалению, для термина делегат не существует подобного разграничения. **И само определение и его порожденный объект** обозначаются одним и тем же словом — "делегат".

Несмотря на то, что делегат происходит от **System.Delegate**, для создания и определения производного делегата не используется привычный синтаксис образования классов (знак двоеточия " **:** "). Вместо этого применяется ключевое слово **delegate**, как показано ниже:

```
public delegate double Calculation(int x, int y);
```

Здесь определен делегат Calculation, который может инкапсулировать любой метод, имеющий два параметра типа **int** и возвращаемое значение метода типа **double**.

В качестве Спецификатора доступности делегат может иметь: **public, protected**, **internal**, **private**

Примечания:

Делегаты являются классами, поэтому их описания можно разместить там же, где и описания обычных классов.

Для делегатов применяются те же спецификаторы доступности, что и для классов. Значение их одинаково.

Показанное ранее определение делегата с именем Calculation **представляет собой определяемый пользователем тип** (как и любое другое определение класса). Этот тип можно использовать для объявления новых переменных аналогичных **Calculation,** например:

Calculation myCalculation;

myCalculation может указывать на любую переменную типа **Calculation.** Новый экземпляр создается обычным способом — при помощи ключевого слова **new** и следующего за ним имени класса делегата, как показано ниже:

myCalculation = new Calculation(<Имя_метода>);

<Имя_метода> представляет метод, который будет **инкапсулирован** в этом экземпляре делегата. Этот метод, как указывалось ранее, должен иметь тот же тип возвращаемого значения (в примере, **double)** и список формальных параметров (два параметра типа **int),** что и указанные в определении делегата. **Имя метода может быть любым**.

myCalculation можно вызвать, как обычный метод:

result = myCalculation (15, 20);

myCalculation делегирует этот вызов методу, который в нем инкапсулирован, вызывая его (с передачей параметров методу). Это и приводит к исполнению содержащихся в нем операторов. Результирующее значение из инкапсулированного метода затем возвращается через **myCalculation** и, в примере случае, присваивается переменной **result**. Все приведенные фрагменты собраны в одну программу (**листинг 1**), раскрывающую устройство делегатов.

```
1  using System;                                          // Делегаты.      Листинг 1,
2
3  namespace ConsAppl_Mic_Ch20_731_Delegate
4  {
5     class Calculator
6     {
7        public delegate double Calculation(int x, int y);              // делегат
8
9        public static double Sum(int num1, int num2)      // 1-й метод на который сылается делегат
10       {
11          return num1 + num2;
12       }
13
14       public static void Main()
15       {
16          double result;
17          Math myMath = new Math();                    // объект класса Math
18
19          Calculation myCalculation = new Calculation(myMath.Average);
          /* экземпляр myCalculation делегата Calculation инкапсулирует метод myMath.Average */
20          result = myCalculation(10, 20);    // опосредованный вызов метода myMath.Average (2-й)
21          Console.WriteLine("\nРезультат использования параметров 10, 20 делегатом
```

```
    myCalculation, инкапсулировавшим метод myMath.Average: {0}", result);
22
23          myCalculation = new Calculation(Sum);
              /* экземпляр myCalculation делегата Calculation инкапсулирует метод Sum */
24          result = myCalculation(10, 20); // опосредованный вызов метода Sum (1-й)
25          Console.WriteLine("\nРезультат использования параметров 10, 20 делегатом
    myCalculation, инкапсулировавшим статический метод Sum: {0}", result);
26          Console.ReadLine();
27      }
28  }
29
30  class Math
41  {
42      public double Average(int number1, int number2) // 2-й метод на который сылается делегат
43      {
44          return (number1 + number2) / 2;
45      }
46  }
47  }
```

Вывод программы

Результат использования параметров **10, 20** делегатом **myCalculation**, инкапсулировавшим метод **myMath.Average**: **15**
Результат использования параметров **10, 20** делегатом **myCalculation**, инкапсулировавшим статический метод **Sum: 30**

Делегат **Calculation** определен в **строке 7**. Два метода — **Sum** (**строки 9-12**) и **Average** (**строки 42-46**) — в программе определены. Их формальные параметры, тип возвращаемого значения совпадают с указанными в делегате **Calculation**. Благодаря этому, они могут быть инкапсулированы экземпляром типа **Calculation**. Следует обратить внимание на то, что имена самих методов (**Sum** и **Average**), а также имена их формальных параметров не совпадают с именами в **Calculation** или **myCalculation**. Компилятор определяет, может ли каждый конкретный делегат инкапсулировать какой-либо метод, не учитывая при этом имен.

В **строке 19** новый экземпляр **Calculation** создается при помощи ключевого слова **new** и передается в качестве аргумента методу **Average** объекта **myMath**. Таким образом, после исполнения **строки 19** **myCalculation** инкапсулирует метод **Average** объекта **myMath**. Ссылаться на любой метод экземпляра, передаваемый делегату как аргумент, следует, указывая и имя объекта, и имя метода, например, **myMath.Average**.

В **строке 20** **myCalculation** делегирует вызов **myMath.Average** (этот метод присвоен строкой раньше). Значит, возвращаемое значение будет равняться среднему арифметическому чисел **10** и **20**.

Строка 23. Создается новый экземпляр **Calculation**, на этот раз инкапсулирующий метод **Sum**. **Sum** имеет атрибут **static** и содержится в классе **Calculation**. Поэтому, при передаче конструктору **Calculator** в качестве аргумента, достаточно указать сам идентификатор — **Sum**.

Вызов **myCalculation** в строке **24** идентичен вызову в строке **20**, но он приведет к другому результату — **myCalculation** при каждом вызове будет инкапсулировать разные методы. Вместо вычисления среднего арифметического, теперь суммируются исходные числа (это, естественно, выполняет метод **Sum**).

СИНТАКСИЧЕСКИЙ БЛОК 1. СОЗДАНИЕ ЭКЗЕМПЛЯРА ДЕЛЕГАТА

экземпляра_делегата = **new Идентификатор_делегата**(<**Метод**>)

Примечания:

<Метод> идентифицирует **метод, который будет инкапсулирован экземпляром делегата**. Конструктору делегата должен передаваться только один аргумент.

Если <Метод> является методом экземпляра другого класса, его нужно указать как **Идентификатор_объекта.Идентификатор_метода**.

Если это просто статический метод другого класса, его указывают как **Идентификатор_класса.Идентификатор_метода**.

1.1. Групповые (многоадресные) делегаты

Каждый из представленных до сих пор делегатов инкапсулировал только один метод. Существуют и групповые делегаты, которые могут инкапсулировать несколько методов. При вызове группового делегата **все инкапсулированные в нем методы вызываются последовательно**. Применение групповых делегатов особенно **полезно при обработке событий**. Например, в программе, управляемой событиями, однократное нажатие на кнопку может привести к последовательному вызову нескольких методов. Конечно, аналогичный результат можно получить при помощи набора делегатов. Но не проще ли воспользоваться групповыми делегатами, которые специально созданы для элегантной обработки событийных ситуаций.

Делегат с типом возвращаемого значения **void** автоматически превращается в групповой. Групповой делегат не может возвращать значений. Пример определения группового делегата **Greeting**:

```
delegate void Greeting();
```

После объявления делегата ему можно присвоить новый экземпляр:

```
Greeting myGreeting = new Greeting(SayThankYou);
```

где объявлен объект **myGreeting** класса **Greeting**. Ему присвоен новый экземпляр **Greeting**, инкапсулирующий метод **SayThankYou**. Аналогично, объявлен и присвоен экземпляр делегату **yourGreeting:**

```
Greeting yourGreeting = new Greeting(SayGoodMorning);
```

Два групповых делегата можно объединить при помощи операции **+** и присвоить результат групповому делегату того же типа. Этот новый делегат будет инкапсулировать все методы обоих делегатов. Например, после выполнения:

```
Greeting ourGreeting = myGreeting + yourGreeting;
```

ourGreeting инкапсулирует все методы, инкапсулированные **myGreeting** (в данном случае, один) и **yourGreeting** (также, один).

В этом же контексте для групповых делегатов можно применять и операцию **+=**:

```
ourGreeting += new Greeting(SayGoodnight);
```

что совпадает с:

```
ourGreeting = ourGreeting + new Greeting(SayGoodnight);
```

В результате в **ourGreeting** к инкапсулированным методам добавляется еще и метод **SayGoodNight**.

По аналогии, для удаления одного группового делегата из другого применяются операции **-** и **- =** .

Например, чтобы удалить **yourGreeting** из **ourGreeting**, применяется операция:

```
ourGreeting = ourGreeting - yourGreeting;
```

что эквивалентно записи:

ourGreeting - = yourGreeting;

Примечание

Операции +, +=, -, -= применимы только к групповым делегатам — делегатам с типом возвращаемого значения **void**.

Почему групповые делегаты должны быть только типа void?

Если делегат объявлен, как возвращающий значение, то метод, который он инкапсулирует, также будет возвращать значение. Когда делегат инкапсулирует только один метод, все работает прекрасно. Значение, возвращаемое инкапсулированным методом, передается через делегат (вернее, его возвращаемое значение) в точку вызова. Естественно, если делегат инкапсулирует более одного метода, то механизма для контроля над обработкой всех этих значений просто не существует. Другими словами, групповые делегаты возвращать значения не могут и обязательно должны быть объявлены как **void**.

2. События

События представляет собой сигнал о том, что в программе произошло нечто, достойное внимания оператора. Примерами событий могут служить нажатия на клавиши, срабатывание таймера, окончание работы принтера. Объект, способный генерировать события, является **источником событий**.

Один или несколько объектов могут **зарегистрироваться**, **на уведомление об определенном событии**, **произошедшем в другом объекте**. Их называют **подписчиками или абонентами**. **Каждый подписчик должен нести в себе метод для обработки события подписки**. Он называется **обработчиком события**. Когда соответствующее событие возникает, один за другим выполняются все обработчики. Подписчик может зарегистрироваться на несколько разных типов событий и содержать в себе несколько различных обработчиков. Для реализации процесса обработки событий групповые делегаты прекрасно приспособлены — именно они формируют связи между подписчиками и издателями.

В чем заключается преимущество деления программ на программы-издатели и программы-подписчики? Почему издатели сами не могут реагировать на генерируемые ими же события? Процесс обработки событий позволяет программистам писать программы с менее тесными связями. Тем, кто пишет объекты-издатели, не всегда обязательно знать об объектах-подписчиках и наоборот. Более того, объекты-подписчики сами могут легко подписываться на конкретные события или отписываться от них во время выполнения, поскольку этот аспект не был жестко запрограммирован в исходном коде.

2.1. Написание программ, управляемых событиями

Событие в C# — специализированный **групповой делегат**, предназначенный для процесса обработки событий. Его легче и надежнее, чем обычные групповые делегаты, использовать для данной цели.

Событие объявляется в программе путем добавления ключевого слова **event** к групповому делегату. Например, если определить следующий групповой делегат **MoveRequest** (значение двух параметров, **sender** и **e,** обсуждается далее):

```
public delegate void MoveRequest(object sender, MoveRequestEventArgs e);
```

а затем вместо объявления обычного группового делегата **MyMoveRequest:**

```
public MoveRequest  MyMoveRequest;
```

добавить ключевое слово **event,** то будет объявлено событие **OnMoveRequest**

```
public event MoveRequest  OnMoveRequest;
```

OnMoveRequest семантически настолько похож на обычный групповой делегат (наподобие **MyMoveRequest),** что фактически в, большинстве приложений, его можно таковым и считать.

Примечание

По принятому соглашению имя любого события начинается с префикса **On**.

Определение делегата и связанное с ним объявление события размещается в классе-издателе события. Обработчик для этого делегата и события размещен в классе-подписчике. Обработчик события должен иметь те же, что и у делегата, формальные параметры и тип возвращаемого значения. Это позволяет обработчику быть инкапсулированным в событии **OnMoveRequest** (при помощи подписки на последнее), а, значит, вызываться при его наступлении.

Примечание

Возбуждение события в C# инициируется его вызовом так же, как и при вызове делегата (см. ранее). В данном случае, это означает выполнение внутри класса-издателя строки наподобие:

```
OnMoveRequest(senderObject, someEventArguments);
```

Обработчик события имеет следующий вид:

```
public void MoveRequestHandler(object sender, MoveRequestEventArgs e)
{
…
}
```

Предположим, что и делегат и, реализованное им событие, определены в классе **GameController**. В настоящее время они находятся в экземпляре **controller.** Подписчики теперь могут подписаться на событие, инкапсулируя его метод **MoveRequestHandler** (**обработчик события**) в событии **OnMoveRequest**, аналогично применению групповых делегатов:

```
controller.OnMoveRequest += new GameController.MoveRequest(MoveRequestHandler);
```

Подобным образом, **MoveRequestHandler** может быть удален из числа подписанных на событие, как показано в следующей строке:

```
controller.OnMoveRequest - = new GameController.MoveRequest(MoveRequestHandler);
```

Примечание

Важное различие между обычным групповым делегатом и событием заключено в том, что вне объекта, в котором оно находится, к событию могут быть применены только операции += и -=. Например, следующий оператор некорректен, поскольку **OnMoveRequest** является событием:

```
controller.OnMoveRequest = new GameController.MoveRequest(MoveRequestHandler); // Неверно
```

Понимание того, что данная строка неверна, свидетельствует о правильности методики. Строка в таком виде могла бы вызвать хаос. Все другие обработчики, инкапсулированные **OnMoveRequest** будут удалены. Вместо них инкапсулируется только приведенный в примере метод **MoveRequestHandler.** Если предположить существование такого оператора, один подписчик мог бы просто удалить всех остальных подписчиков.

Когда компилятор встречает в программе ключевое слово **event,** он явно объявляет это событие закрытым **(private)** и создает два открытых **(public)** свойства (видны только в MSIL) с именами **add_<Имя_события>** и **remove_<Имя_события**>, доступ к которым осуществляется операциями += и -=.

Вернемся теперь к параметрам делегата **MoveRequest** (**object sender, MoveRequestEventArgs e**) ,

при помощи которого реализовано событие. По общему соглашению, принятому **Microsoft**, делегаты, с помощью которых реализованы события, всегда имеют два параметра. Первый — типа **System.Object** (в C# это псевдоним **object**). И второй — типа **System.EventArgs** (или его подкласса), как показано далее (где **MoveRequestEventArgs** — подкласс **System.EventArgs**):

```
public delegate void MoveRequest(object sender, MoveRequestEventArgs e);
```

Параметр **object** позволяет объекту-издателю информировать обработчик события подписавшегося объекта, где (в каком именно объекте) возникло событие (поэтому он часто называется **sender**). Событие, как мы выяснили ранее, возбуждается самим объектом-издателем. Ссылка на данный объект находится в ключевом слове **this**. Оно, в свою очередь, выступает первым аргументом, как показано в следующем выражении:

```
OnMoveRequest(this, new MoveRequestEventArgs(MoveRequestType.FastForward));
```

Второй параметр, **MoveRequestEventArgs** (подкласс **System.EventArgs**), позволяет объекту-издателю передать подписчику подробную информацию о событии.

Резюме

Рассмотрели темы делегатов, событий, а также их значение для разработки на языке C# управляемых событиями программ.

Делегат является подклассом **System.Delegate**. **Он определяет тип возвращаемого значения и параметры для методов, которые он может инкапсулировать**. **Делегат при вызове передает управление методу, который он инкапсулирует**.

Делегаты позволяют отсрочить решения о реализации методов до времени исполнения программы. Например, массив делегатов позволяет определить последовательность операций во время исполнения, а реализации методов могут быть переданы в качестве аргументов.

Групповой делегат может инкапсулировать несколько методов. Любой делегат с типом возвращаемого значения **void** автоматически является групповым. Такие делегаты особенно полезны при использовании в программах, управляемых событиями.

Событие сигнализирует о том, что в программе произошло нечто, заслуживающее внимания. В C# события реализованы при помощи групповых делегатов. Управляемая событиями программа состоит из **объектов-издателей, генерирующих события**, и **объектов-подписчиков, содержащих обработчики**, которые и реагируют на эти события.

Контрольные вопросы

1. Что общего и в чем различия в применении делегатов и абстрактных методов?
2. Почему делегат — удачное название для этой конструкции?
3. В каком месте программы можно разместить определение делегата?

Рассмотрим определение делегата:

```
public delegate int Filtering(string str);
```

4. Какие из нижеследующих методов могут инкапсулировать экземпляр этого делегата?

а. `protected int Filtering(string myString, double x) {...}`

б. `internal static int FilteringOp(string myStr) {...}` **// верно**

в. `public double Filtering(string str) {...}`

г. `public short Sum(int x, int y) {...}`

5. Почему все групповые делегаты принадлежат типу **void**?

6. **а)** Какие арифметические операции можно использовать при работе с групповыми делегатами?

б) Какие арифметические операции можно использовать при работе с событиями, если вызов идет извне пределов объекта, которому события принадлежат?

7. Что такое обработчик события?

Упражнения по программированию

Добавьте статический метод **Product** в класс **Math** из листинга 1. Этот метод должен вычислять произведение двух аргументов и быть инкапсулирован экземплярами делегата **Calculation**. Напишите код с использованием делегата для проверки этого метода.

Ответы на контрольные вопросы

1. Сходства:

• Они задают тип возвращаемого значения и параметры метода, который (во время исполнения) они могут представлять и вызывать.

• Они позволяют отложить решение о методе вызова до самого момента исполне
ния программы.

Различия:

• Имя метода должно совпадать с именем метода в абстрактном классе. Для делегата имя не имеет значения.

• Абстрактные методы используют при работе такие механизмы: наследование, интерфейсы, динамическое связывание.

• Делегаты лучше приспособлены для обработки событий.

2. Делегат вызывается так же, как метод, но, в отличие от метода, не исполняет вызов самостоятельно. Вместо этого он делегирует исполнение инкапсулированному в нем методу.

3. Делегат представляет собой класс, поэтому его определение можно разместить там, где и определение класса.

4. б.

5. Групповой делегат может инкапсулировать несколько методов. Если его тип будет отличаться от void, все инкапсулируемые методы будут возвращать значение. Синтаксис языка не позволяет указать, как именно должны обрабатываться эти значения.

6. **а.** +, -, +=, -=

б. +=, -=

7. Обработчик события — это метод, содержащийся в объекте, который на него подписался. Он запускается, когда объект-издатель генерирует это событие.

Ответы на упражнения по программированию

Упражнение 1: Вставьте в класс **Math** следующий метод:

```
public static double Product(int number1, int number2)
{
return (number1 * number2);
}
```

Для проверки этого метода добавьте в метода **Main** следующие строки:

```
myCalculation = new Calculation(Math.Product);
result = myCalculation(10, 20);
Console.WriteLine(\nРезультат использования параметров 10, 20 делегатом myCalculation,
инкапсулировавшим метод Math.Product: { 0}", result);
```