Делегаты. События

Делегаты

- Делегат – это так называемый “безопасный указатель на функцию”,
- делегаты C# могут вызывать более одной функции (при совместном комбинировании двух делегатов результатом будет делегат, который вызывает их обоих).
- “безопасность” – в C++ указатель на функцию – это фактически просто адрес, делегат позволяет проверять количество передаваемых параметров, возвращаемое значение и т.д.
- Многие конструкции C# - это классы, так и делегат - это класс отнаследованный от базового класса System.MulticastDelegate.
- Делегат можно объявлять как в классе, так и просто в пространстве имен.

Создание делегата

1. объявляется сам делегат как тип и сразу же задается прототип функций, которые в будущем могут адресоваться экземплярами этого типа.
2. объявляется ссылка типа делегата, которая будет указывать на коллекцию экземпляров делегата.
3. создаются сами объекты делегата, каждый из которых адресует одну функцию, и комбинируются в коллекцию, на которую указывает объявленная ссылка.

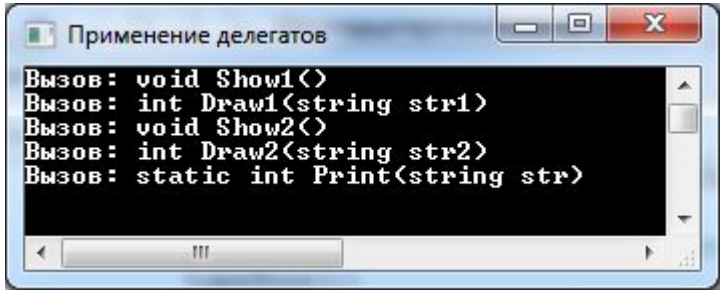
- Коллекция - собой список адресуемых делегатом функций и поддерживается операциями += и -= или статическими методами `Delegate.Combine()` и `Delegate.Remove()`
- Экземпляр делегата способен вызывать эти методы либо по одному, либо сразу весь список.

Одноадресная работа делегатов

- Во время выполнения программы один и тот же делегат можно использовать для вызова разных методов одинаковой сигнатуры простой заменой метода, на который ссылается делегат (главное достоинство делегата, что адресуемый им метод определяется не на этапе компиляции, а в период выполнения). Заменяя в делегате адресуемые функции мы обеспечим их полиморфный вызов.

- Область видимости делегата можно регулировать точно так же, как и область видимости любого другого типа в приложении.

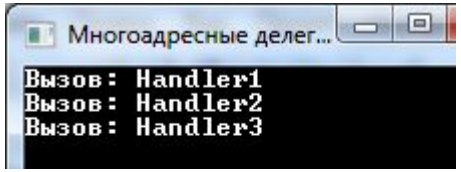
```
class DelegateTest      {
        // Объявляем функции с сигнатурой делегатов
        public void Show1()         {
            Console.WriteLine("Вызов: void Show1()");         }
        // Объявляем функции с сигнатурой делегатов
        public void Show2()         {
            Console.WriteLine("Вызов: void Show2()");         }
        // Объявляем функции с сигнатурой делегатов
        public int Draw1(string str1)         {
            Console.WriteLine("Вызов: {0}", str1);   return 1;   }
        // Объявляем функции с сигнатурой делегатов
        public int Draw2(string str2)         {
            Console.WriteLine("Вызов: {0}", str2); return 2;    }
        // Объявляем статическую функцию
        public static int Print(string str)         {
            Console.WriteLine("Вызов: {0}", str);   return 0;   } }
    // Объявляем делегат в пространстве имен
    delegate void TypeShow();
    // Вызывающая сторона
    class MyClass      {
        // Объявляем делегат в классе
        delegate int TypeDraw(string str);
        public MyClass()         {
            // Создаем экземпляр класса с методами
            DelegateTest delegateTest = new DelegateTest();
            // Объявляем ссылки на объекты делегатов
            TypeShow typeShow;
            TypeDraw typeDraw;
            // Создаем объекты делегатов
            typeShow = new TypeShow(delegateTest.Show1);
            typeDraw = new TypeDraw(delegateTest.Draw1);
            // Вызываем методы посредством делегатов
            typeShow();
```



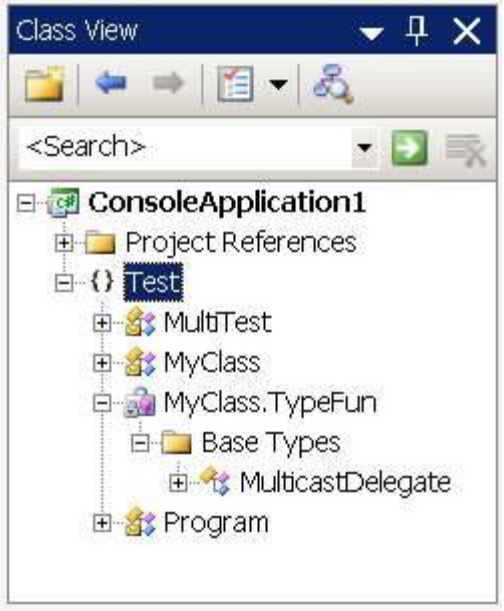
- Делегат может адресовать как методы объекта (экземпляра класса), так и статические методы.
- Надо, чтобы заголовки объявления делегата и объявления метода совпадали как по типу возвращаемого значения, так и по сигнатуре.

Многоадресная работа делегатов

```
// Для методов, имеющих одинаковую сигнатуру и не возвращающих значение (только
    с void) с помощью делегата можно организовать сразу цепочку вызовов
class MultiTest   {
    // Статическое поле
    static int x = 1;
    // Объявляем функции с одинаковой сигнатурой
    public void Handler1(string name)   {
      Console.WriteLine(name + x++);      }
    public void Handler2(string name)   {
      Console.WriteLine(name + x++);      }
    public void Handler3(string name)  {
      Console.WriteLine(name + x++);      }   }
  // Вызывающая сторона
  class MyClass   {
    // Объявляем делегат в классе
    delegate void TypeHandler(string text);
    public MyClass()      {
      // Создаем экземпляр класса с методами
      MultiTest obj = new MultiTest();
      // Создаем объект-делегат и заполняем адресами
      TypeHandler Handler = new TypeHandler(obj.Handler1);
      Handler += new TypeHandler(obj.Handler2);
      Handler += obj.Handler3;   // Упрощенный синтаксис
      // Вызываем цепочку методов
```



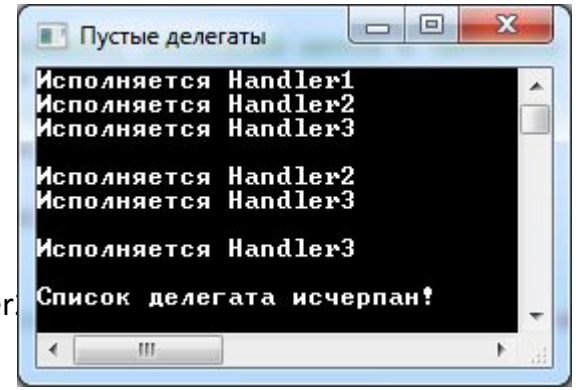
- Делегаты являются экземплярами типа System.MulticastDelegate, который в свою очередь наследует абстрактный класс System.Delegate.
- Экземпляр типа **MulticastDelegate** может хранить в себе одну или сразу несколько ссылок на методы. В любом случае мы не можем явно объявлять делегат с помощью типа **MulticastDelegate**.
- Это делается с помощью ключевого слова **delegate**, но неявно порождается объект класса **MulticastDelegate**.
- структура примера через панель **Class View**.



Пустые делегаты

- Делегат существует как объект до тех пор, пока его список с адресуемыми функциями содержит хотя-бы одну функцию.
- Как только список делегата становится пустым, то он превращается в обычную ссылочную переменную с нулевой адресацией.
- Т.к. мы можем не только пополнять список, но и удалять из него некоторые функции, то необходимо следить за тем, чтобы список делегата не оказался пустым, иначе при попытке вызова такого списка будет выброшено исключение.

```
class MultiTest   {
      // Объявляем функции одного прототипа
      public void Handler1()      {
         Console.WriteLine("Исполняется Handler1");      }
      public void Handler2()      {
         Console.WriteLine("Исполняется Handler2");      }
      public void Handler3()      {
         Console.WriteLine("Исполняется Handler3");      }   }

   // Вызывающая сторона
   class MyClass   {
      // Для заголовка консольного окна
      public static string Title = "Пустые делегаты“;
      // Объявляем делегат как член класса
      delegate void MyDelegate();
      public MyClass()      {
         // Создаем экземпляр класса с методами
         MultiTest obj = new MultiTest();
         // Создаем объект-делегат и заполняем ссылками на функции
         // Инициализируем первой ссылкой
         MyDelegate del = new MyDelegate(obj.Handler1);
         // Добавляем другие ссылки
         del += obj.Handler2;
         del += obj.Handler3;
         // Вызываем цепочку методов
         del();
         Console.WriteLine();
         // Извлекаем метод и опять вызываем цепочку методов
         del -= obj.Handler1;
         del();
         Console.WriteLine();
         // Извлекаем метод и опять вызываем цепочку методов
         del = (MyDelegate)Delegate.Remove(del, new MyDelegate(obj.Handler
         del();
         Console.WriteLine();
         // Извлекаем последний метод и пытаемся адресоваться к пустому делегату
```



- если делегат имеет пустой список, т.е. не ссылается ни на одну функцию, то представляющая его ссылка имеет нулевой адрес

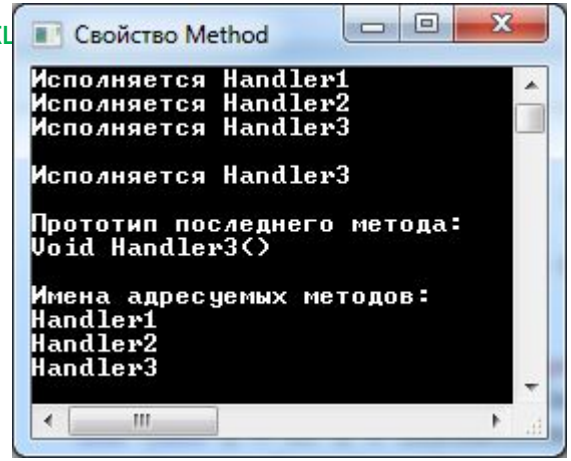
Некоторые члены типа MulticastDelegate

- Тип **MulticastDelegate** сам имеет несколько своих собственных членов, но большую часть он наследует от абстрактного класса **Delegate**.
- Члены типа **MulticastDelegate** содержат все необходимые сервисы по управлению делегатами. Некоторые из этих членов экземплярные, а некоторые - статические.

Некоторые члены типа MulticastDelegate, унаследованные от Delegate

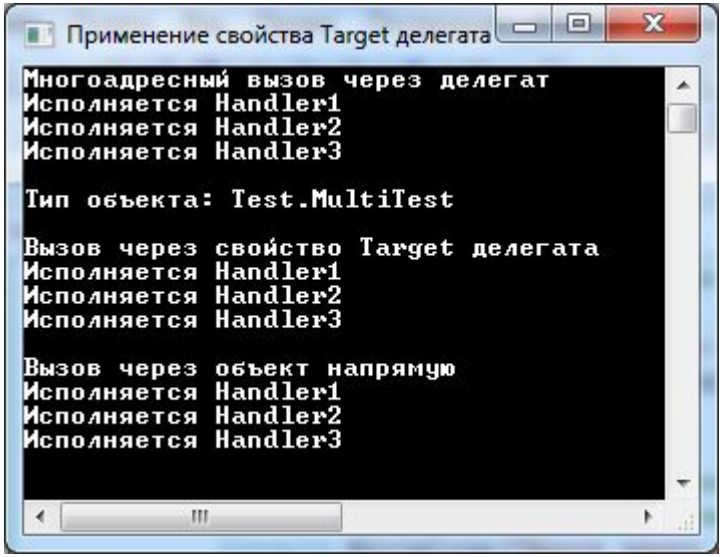
Член	Пояснения
`Method { get; }`	Возвращает последнюю добавленную в делегат ссылку на метод
`Target { get; }`	Возвращает ссылку на объект, связанный с адресуемым делегатом экземплярным методом
`DynamicInvoke()`	Позволяет динамически обратиться к функциям, привязанным к делегату
`GetInvocationList()`	Возвращает список функций, привязанных к делегату
`operator ==`	Перегруженный оператор для сравнения содержимого делегатов
`operator !=`	Перегруженный оператор, позволяющий определить различие содержимого делегатов
`Combine()`	Добавляет в объект делегата новую ссылку на функцию
`Remove()`	Удаляет из объекта делегата ссылку на указанную функцию
`CreateDelegate()`	Динамически создает делегат

```
//Применение свойства Method
class MultiTest  {
    // Статическое поле
    static int x = 1;
    // Объявляем функции с одинаковой сигнатурой
    public void Handler1()     {
      Console.WriteLine("Исполняется Handler1");     }
    public void Handler2()     {
      Console.WriteLine("Исполняется Handler2");     }
    public void Handler3()     {
      Console.WriteLine("Исполняется Handler3");     }
  }

  // Вызывающая сторона
  class MyClass  {
    public static string Title = "Свойство Method“;
    // Объявляем делегат как член класса
    delegate void MyDelegate();
    public MyClass()     {
      // Создаем экземпляр класса с методами
      MultiTest obj = new MultiTest();
      // Создаем объект-делегат и заполняем ссылками на функц
      // Инициализируем первой ссылкой
      MyDelegate del = new MyDelegate(obj.Handler1);
      // Добавляем другие ссылки
      del += obj.Handler2;
      del += obj.Handler3;
      // Вызываем цепочку методов
      del();
      Console.WriteLine();
      del.Method.Invoke(obj, null); // Вызываем последний метод
      Console.WriteLine("\nПрототип последнего метода:\n{0}",
        del.Method.ToString());
      Console.WriteLine("\nИмена адресуемых методов:");
      Delegate[] listMethods = del.GetInvocationList();
      for (int i = 0; i < listMethods.Length; i++)
```



```
//Свойство Target - свойство возвращает объект, связанный с адресуемым делегатом методом
class MultiTest    {
      // Объявляем функции с одинаковой сигнатурой
      public void Handler1()       {
         Console.WriteLine("Исполняется Handler1");       }
         public void Handler2()       {
         Console.WriteLine("Исполняется Handler2");       }
       public void Handler3()       {
         Console.WriteLine("Исполняется Handler3");       }    }
      // Вызывающая сторона
   class MyClass    {
      public static string Title = "Применение свойства Target делегата";
         // Объявляем делегат как член класса
      delegate void MyDelegate();
      public MyClass()       {
         // Создаем экземпляр класса с методами
         MultiTest obj = new MultiTest();
// Создаем объект-делегат и заполняем ссылками на функции
         // Инициализируем первой ссылкой
         MyDelegate del = new MyDelegate(obj.Handler1);
         // Добавляем другие ссылки
         del += obj.Handler2;       del += obj.Handler3;

         // Вызываем цепочку методов
         Console.WriteLine("Многоадресный вызов через делегат");
         del();
         Console.WriteLine();
```

```
// Запуск
   class Program    {
      static void Main()       {
         // Настройка консоли
         Console.Title = MyClass.Title;
      new MyClass();   // Исполняем
   }    }
```

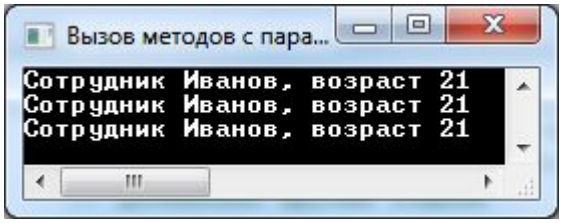


Методы DynamicInvoke() и GetInvocationList()

- позволяет вызывать отдельные члены, адресуемые списком объекта-делегата, и задавать требуемые аргументы.
- Если член списка не имеет аргументов, то в качестве параметра метода используется **null**, иначе - массив параметров адресуемого члена.
- объявление ссылки на объект-делегат прописывает только прототип методов, которые корректно может адресовать эта ссылка.
- делегат как объект создается при заполнении списка адресуемыми функциями. Но при формировании списка в него добавляются только имена методов. Это приемлемо, если вызываются функции с пустой сигнатурой.

```
class ShowPerson    {
      // Объявляем функцию с аргументами
      public static void Handler(string name, int age)       {
         Console.WriteLine("Сотрудник {0}, возраст {1}", name, age);       }    }
   // Вызывающая сторона
   class MyClass    {
      public static string Title = "Вызов методов с параметрами";
      // Объявляем делегат как член класса
      delegate void MyDelegate(string name, int age);
      public MyClass()       {
         // Создаем и заполняем объект-делегат
         MyDelegate del = new MyDelegate(ShowPerson.Handler);
         // Добавляем другие ссылки
         int count = 3;
         for (int i = 1; i < count; i++)          {
            del += ShowPerson.Handler;          }
         // Вызываем цепочку методов с одинаковым параметром
         del("Иванов", 21);       }    }

   // Запуск
   class Program    {
      static void Main()       {
         // Настройка консоли
         Console.Title = MyClass.Title;
         new MyClass();    // Исполняем
   }  }
```

все зарегистрированные в делегате методы стороннего класса получают, при вызове их с помощью делегата, одинаковую входную информацию



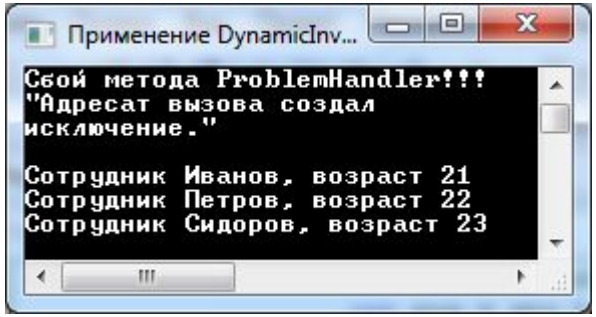
- если с помощью делегата нужно адресовать методы, имеющие разные значения параметров используется

public object DynamicInvoke(params object[ ] args)

- с помощью метода **DynamicInvoke()** можно решить проблему **надежности кода** при адресации вызовов методов посредством делегата.
- Если любой из цепочки метод при традиционном вызове может дать сбой, то в результате система выдаст исключение и прервет вызовы остальных методов списка. Исключение мы можем обработать, но только для списка в целом, если не будем контролировать индивидуально каждый член списка делегата.
- Эти проблемы решает метод **DynamicInvoke()** совместно с **GetInvocationList()**

```
class ShowPerson    {
      // Функция с аргументами
      public static void Handler(string name, int age)       {
         Console.WriteLine("Сотрудник {0}, возраст {1}", name, age);   }
      // Проблемная функция с нормальным прототипом
      public static void ProblemHandler(string name, int age) {
         // Преднамеренно выбрасываем исключение
         throw new Exception();       }    }
   // Вызывающая сторона
   class MyClass    {
      public static string Title = "Применение DynamicInvoke()“;
      // Объявляем делегат
      delegate void MyDelegate(string name, int age);
      public MyClass()       {
// Формируем список объекта-делегата
// Добавляем в список один проблемный метод
         MyDelegate del = new MyDelegate(ShowPerson.ProblemHandler);
         // Добавляем еще три нормальных метода
         int count = 3;
         for (int i = 0; i < count; i++)
         {              del += ShowPerson.Handler;          }
         object[] param = new object[2];   // Объявили массив для параметров
         int j = 0;    // Объявили и инициализировали счетчик
         // Перебираем список вызовов делегата, включая и вызов проблемного метода
         foreach (Delegate d in del.GetInvocationList())          {
            // Индивидуально формируем параметры методов
            switch (j)      {
               case 0:   // Можно и не задавать, все равно для проблемного метода!
                  param[0] = "Мистер X";
                  param[1] = 99;                    break;
               // Для вызовов нормального метода
```

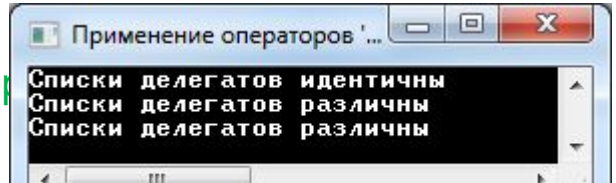
```
// Запуск
  class Program
  {
     static void Main()
     {
        // Настройка консоли
        Console.Title = MyClass.Title;
        new MyClass();   // Исполняем
}
  }
```



Перегруженные операторы 'operator ==' и 'operator !='

- Эти операторы позволяют подтвердить или опровергнуть абсолютную идентичность списков функций сравниваемых делегатов.

```
class Handler    {
      // Функции
      public void Handler1()    {       }
         public void Handler2()       {          ;       }
   }
      // Вызывающая сторона
   class MyClass    {
      public static string Title = "Применение операторов '==' и '!='";
         // Объявляем делегат
      delegate void MyDelegate();
      // Объявляем ссылки на делегаты как
      // поля для видимости в методах класса
      MyDelegate del0, del1;
      public MyClass()       {
         // Создаем объект
         Handler obj = new Handler();
         // Формируем список вызовов объекта-делегата
         del0 = new MyDelegate(obj.Handler1);
         del0 += new MyDelegate(obj.Handler2);
         // Еще один делегат с тем же списком вызовов
         del1 = new MyDelegate(obj.Handler1);
         del1 += obj.Handler2;       // Упрощенный синтаксис
            // Сравниваем делегаты с полностью совпадающими списками
         Compare();
            // Делегат прежним содержимым, но в другом порядке
         del1 = new MyDelegate(obj.Handler2);
         del1 += obj.Handler1;   // Упрощенный синтаксис
            // Сравниваем делегаты с одинаковым содержимым, но
         Compare();
            // Изменяем содержимое одного из делегатов
```



Методы Combine() и Remove()

- Это статические методы, способные получать новый делегат как объединение списков двух делегатов одного и того же типа, или получать новый делегат с усеченным списком при тех же условиях.
- Методы могут использоваться вместо перегруженных операций '+=' или '-=' при последовательном изменении списка одиночными функциями по синтаксису

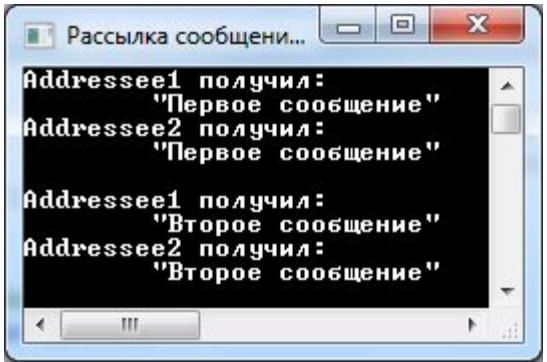
```
MyDelegate del;
del = new MyDelegate(obj.Handler1);
del += new MyDelegate(obj.Handler2);
// Или del = new MyDelegate(obj.Handler1);
del = (MyDelegate)Delegate.Combine(del, new
MyDelegate(obj.Handler2));
```

```
class Handler    {
      // Функции
      public void Handler1()      {
         Console.WriteLine("Вызов Handler1()");      }
      public void Handler2()      {
         Console.WriteLine("Вызов Handler2()");      }
   }
   // Вызывающая сторона
   class MyClass    {
      public static string Title = "Применение Combine() и Remove()“;
      // Объявляем делегат как член класса
      delegate void MyDelegate();
      public MyClass()      {
         // Создаем объект
         Handler obj = new Handler();
         // Формируем список объекта-делегата из 2 вызовов
         MyDelegate del1 = new     MyDelegate(obj.Handler1);
         del1 += new MyDelegate(obj.Handler1);
         // Еще один делегат того же типа из 3 вызовов
         MyDelegate del2 = new MyDelegate(obj.Handler2);
         del2 += obj.Handler2;   // Упрощенный синтаксис
         del2 = del2 + obj.Handler2; // То же самое
         // Новый делегат из 5 вызовов
         MyDelegate del3 = (MyDelegate)Delegate.Combine(del1,
         // Вызываем 5 функций
         del3();
         Console.WriteLine();
         // Вновь формируем делегаты
         del1 = new MyDelegate(obj.Handler1);
         del1 += obj.Handler2;
```

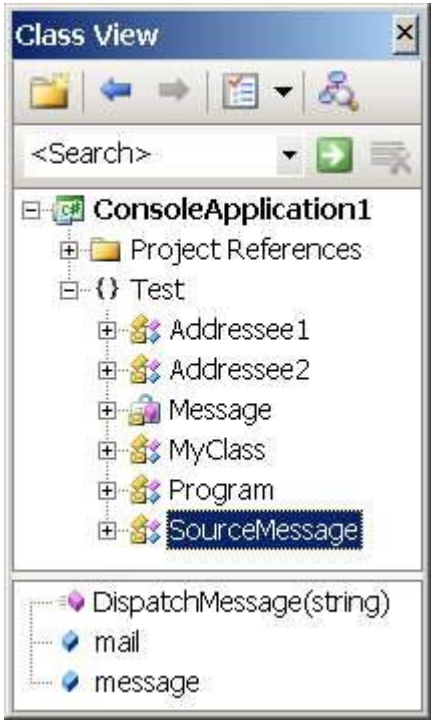
События в C#
Рассылка сообщений с помощью делегата

- Чтобы любые объекты могли обмениваться информацией, они должны разговаривать на одном языке. В случае объектно-ориентированного программирования таким условием является одинаковый прототип функций, определяемый делегатом. Многоадресная работа делегата удобна тем, что можно послать одинаковое сообщение сразу нескольким объектам, функции которых зарегистрированы в списке делегата как обработчики этого сообщения. Список делегата можно назвать по-разному:
- список вызываемых функций,
- список обработчиков,
- список адресатов,
- список получателей и т.д.
- Напомним, что если список адресатов пуст, то самого объекта-делегата не существует и ссылка на него имеет значение **null**. Без проверки этого обстоятельства при попытке вызова адресатов пустой ссылкой-делегатом будет сгенерировано стандартное исключение**NullReferenceException**.

```
// Образец сообщения определяется делегатом
   delegate void Message(string message);
   // Источник сообщения
   class SourceMessage    {
      // Общедоступное поле ссылки на объект-делегат,
      // который наполнится указателями
      // на функции в классах-получателях
      public Message mail;
      // Необязательное поле с рассылаемым сообщением
      public string message;
       // Разослать сообщение - функция диспетчеризации
      public void DispatchMessage(string mess)       {
         // Сохраняем внешнее сообщение во внутреннем поле
         message = mess;
         // Инициируем рассылку сообщения всем,
         // кто зарегистрировался в объекте-делегате
         if (mail != null)   // Если не пустой делегат
            mail(mess);       }    }
      // Получатель сообщения
   class Addressee1    {
      // Функции
      public void Handler(string message)       {
         Console.WriteLine("Addressee1 получил:"
            + "\n\t\"{0}\"", message);       }    }
     // Получатель сообщения
   class Addressee2    {
      // Функции
      public void Handler(string message)       {
         Console.WriteLine("Addressee2 получил:"
            + "\n\t\"{0}\"", message);       }    }
    // Вызывающая сторона
   class MyClass    {
      static public string Title = "Рассылка сообщений делегатом";
         public MyClass()       {
         // Создаем объекты источника и получателей сообщения
```



- В классе-источнике сообщения **SourceMessage** мы объявили два общедоступных поля: ссылку **mail** на объект-делегат и переменную **message** для хранения сообщения внутри класса. Поле-переменная **message** для работы программы не нужна, но мы ее ввели просто для того, чтобы сравнить представления в панели **Class View**.



Превращение делегата в событие

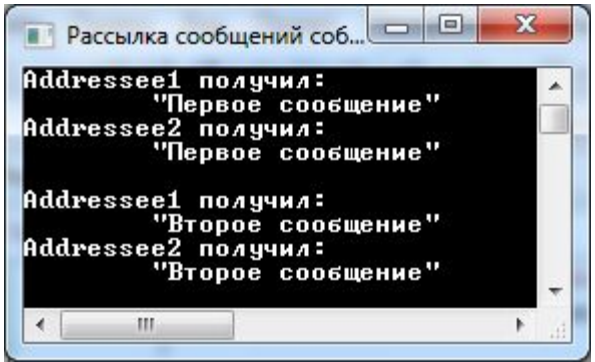
- Разработчики **C#** и библиотеки **.NET Framework** решили объявлять ссылку на объект делегата, когда он используется для рассылки сообщений, с ключевым словом **event** и называть ее **событием**.

Особенность:

- событие-делегат является собственностью класса-источника и вызывать его напрямую из класса-приемника как поле сообщений нельзя.
- Событие должно инициироваться функцией-членом класса, отсылающего сообщение (**функция диспетчеризации**), в котором оно для этих целей и объявлено.
- В приемнике сообщения можно только подписаться на событие источника путем прикрепления нужных функций-обработчиков, а далее вызвать функцию диспетчеризации владельца события.
- Наполнение списка события конкретными адресами функций выполняют объекты-делегаты.
- Объявление в классе-источнике делегата как события означает только возможность на него подписаться функциям внешнего кода, но не гарантирует, что на него действительно кто-то подпишется в момент возбуждения этого события в классе-источнике. Поэтому функция диспетчеризации вначале проверяет, не пустой ли делегат, и только потом выполняет вызов связанных с ним методов-обработчиков.

```
// Образец сообщения определяется делегатом
delegate void Message(string message);
  // Источник сообщения
class SourceMessage   {
   // Общедоступное поле ссылки на объект-делегат,
   // который теперь называется событием и наполняется
   // указателями на функции в классах-получателях
   public event Message mail;
   // Необязательное поле с рассылаемым сообщение
   public string message;
   // Разослать сообщение - функция диспетчеризации
   public void DispatchMessage(string mess)      {
// Сохраняем внешнее сообщение во внутреннем поле
      message = mess;
      // Инициируем рассылку сообщения всем,
      // кто зарегистрировался в объекте-делегате
      if (mail != null)  // Если не пустой делегат
         mail(mess);      }   }
   // Получатель сообщения
class Addressee1   {
   // Функции
   public void Handler(string message)      {
      Console.WriteLine("Addressee1 получил:"
         + "\n\t\"{0}\"", message);      }   }
  // Получатель сообщения
class Addressee2   {
   // Функции
   public void Handler(string message)      {
      Console.WriteLine("Addressee2 получил:"
         + "\n\t\"{0}\"", message);      }   }
   // Вызывающая сторона
class MyClass   {
   static public string Title = "Рассылка сообщений событием";
   public MyClass()      {
// Создаем объекты источника и получателей сообщения
```

функция диспетчеризации **DispatchMessage()**, имитирующее событие



- Когда вместо объекта-делегата для передачи сообщений используется объект-событие, то включенные в список события функции называют **обработчиками**.
- Клиенты объекта с событием могут подписаться на его обработку.
- При возникновении события обработчики клиентов будут выполняться в том порядке, в котором они следуют в списке события.
- Сбой любого обработчика прервет выполнение оставшихся членов списка события(как и в случае с делегатами)

Типичный способ создания событий

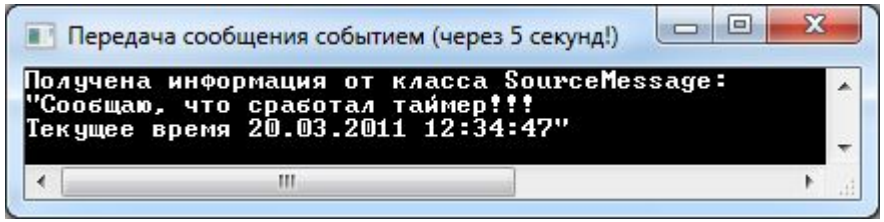
- В соответствии с теорией объектно-ориентированного программирования событие должно передаваться объектом-источником при возникновении определенных обстоятельств.
- Другие классы, содержащие в себе объект с общедоступным полем-событием, по отношению к этому событию могут выступать только как клиенты (внешний код). Они могут только подписаться традиционным способом на событие и прослушивать его, а при получении - реагировать кодом прикрепленных к событию **своих** обработчиков.

```
// Образец сообщения определяется делегатом
   delegate void Message(object sender, string message);
   // Класс-источник сообщения
   class SourceMessage    {
      // Общедоступное поле ссылки на событие
      public event Message Mail;
      // Метод диспетчеризации события, объявили виртуальным
      // и защищенным для возможности замещения в наследниках
      protected virtual void OnMail(string mess)       {
         // Инициируем рассылку сообщения всем,
         // кто подписался на событие
         if (Mail != null)      // Если не пустой делегат
            Mail(this, mess);   // Инициируем событие
      }
      // Объявляем и инициируем внутреннее поле базовым сообщением
      string message = "Сообщаю, что сработал таймер!!!\n”  + "Текущее время ";
      // Объявляем внутреннее поле для видимости в методах
      System.Timers.Timer timer;
      // Конструктор класса-источника сообщения
      public SourceMessage()       {
         // Создаем и запускаем таймер, который по истечении
         // заданного времени инициирует рассылку сообщения
         timer = new System.Timers.Timer();
         timer.Interval = 5000d; // Сработает через 5 секунд
         timer.Elapsed += new System.Timers.ElapsedEventHandler(timer_Elapsed);
         timer.Start();      // Запускаем таймер
      }
      // Инициирование события из внешнего источника
      void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)       {
         // Извлекаем текущее системное время
         DateTime curTime = DateTime.Now;
```

в классе-источнике сообщения мы объявили поле-ссылку на событие как общедоступную, чтобы в любом классе-клиенте, содержащем объект с событием, эта ссылка была видна и позволяла подписать обработчики

Для инициации события мы в качестве внешней причины применили срабатывание системного таймера. В реальных условиях это может быть что угодно, но чаще всего события генерируются операционной системы, как реакция на действия пользователя или как ответ на переход программных объектов в определенные состояния.

```
        // Дополняем базовое сообщение временем
        message += curTime.ToString();
        OnMail(message);    // Вызываем метод диспетчеризации
        timer.Stop();       // Останавливаем системный таймер
    }   }

  // Класс-получатель сообщения
  class MyClass    {
    static public string Title =
      "Передача сообщения событием (через 5 секунд!)";
    public MyClass()       {
      // Создаем объект с событием
      SourceMessage obj = new SourceMessage();
      // Подписываем на событие обработчик,
      // печатающий полученную информацию,
      // с помощью объекта-делегата
      obj.Mail += new Message(obj_Mail);
    }
    // Обработчик. Вызывается автоматически при возникновении причины
    void obj_Mail(object sender, string message)       {
      String str = "Получена информация от класса “   + sender.GetType().Name + ":\n“  + "\"" + message + "\"";
      Console.WriteLine(str);
    }   }

  // Запуск
  class Program    {
    static void Main()       {
      // Настройка консоли
      Console.Title = MyClass.Title;
      new MyClass();  // Исполняем
  }
}
```



- Модель программирования, основанная на событиях, сейчас является наиболее популярной, а для некоторых задач - и единственно возможной.
- Она наиболее приспособлена для построения интерактивных приложений, где требуется организовать диалог с пользователем.
- Когда пользователь не производит никаких действий, приложение находится в состоянии простоя (**idle** - простой) и требует минимальных ресурсов компьютера.

Создание событий с контролем адресатов

- Используя библиотечные классы, содержащие события:
  - Подписываем в клиенте обработчики на уже готовые события

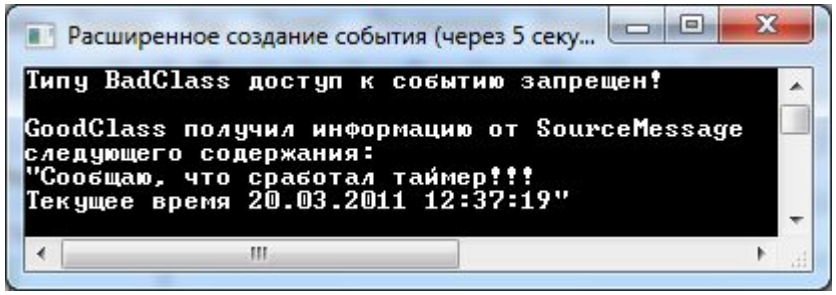
- Влияние на событие библиотечного класса:
  - событие объявляется общедоступными и может наследоваться как обычный член класса(Это дает право расширить библиотечный класс, в котором можно выполнить какие-то действия: скрыть событие, объявив его заново с ключевым словом **new** как закрытое, или переопределить диспетчер события.)
  - Диспетчер в библиотечных классах обычно объявляется виртуальным и может быть переопределен в производном классе или даже совсем скрыт для следующих потомков его переобъявлением.

- Для решения подобных задач существует другой способ создания события в классе-источнике, который является более гибким и его называют **расширенным**. Этот способ позволяет отслеживать в классе, содержащем событие, те обработчики, которые клиентский код будет присоединять к нему.
- Создание события расширенным способом :
  - вместо базового поля используется закрытое поле-делегат
  - используются ключевые слова **add** и **remove**,
  - методы **add** и **remove** автоматически вызываются при добавлении обработчика в список делегата или удаления его из списка.

```
// Объявление делегата как типа
   delegate void Message(object sender, string message);
   // Класс-источник сообщения
   class SourceMessage    {
// Это был стандартный способ
// public event Message Mail;
// Это расширенный способ
// Объявление внутреннего поля как экземпляра делегата
       private Message mail;
          // Создание события с контролем адресатов
       public event Message Mail       {
          add     // Контролируем добавление обработчиков в список
          {
             // Имя получателя сообщения
             string targetName = value.Target.GetType().Name;
             // Выявляем того, кого не любим!
             if (targetName == "BadClass")
                // Ласково уведомляем
                Console.WriteLine("Типу BadClass доступ к событию запрещен!\n");
             else                mail += value;          }
          remove  // Удаление контролировать не будем.
        //"Его там не стояло!"
          {             mail -= value;          }       }
      // Все остальное то-же самое, кроме замены
      // события Mail на закрытое поле-делегат mail
      // Метод диспетчеризации события. Объявили виртуальным
      // и защищенным для возможности замещения в наследниках
      protected virtual void OnMail(string mess)       {
         // Инициируем рассылку сообщения всем,
         // кто подписался на событие
         // Здесь используется поле-делегат, внутри можно (и нужно!)
```

```
      OnMail(message);    // Вызываем метод диспетчеризации
      timer.Stop();       // Останавливаем системный таймер
   }   }
// Хороший класс-получатель сообщения
class GoodClass    {
   public GoodClass()       {
      // Создаем объект с событием
      SourceMessage obj = new SourceMessage();
      // Подписываем на событие обработчик,
      // печатающий полученную информацию
      obj.Mail += new Message(obj_Mail);       }
   // Обработчик
   void obj_Mail(object sender, string message)       {
      String str = this.GetType().Name
         + " получил информацию от "
         + sender.GetType().Name + "\n"
         + "следующего содержания:\n\""
 + message + "\"";
      Console.WriteLine(str);       }   }
  // Плохой класс, которому не разрешено получать сообщения
class BadClass    {
   public BadClass()       {
      // Создаем объект с событием
      SourceMessage obj = new SourceMessage();
  // Подписываемся на событие в неведении,
      // что мы попали в черный список и нас
      // не включат в список рассылки
      obj.Mail += new Message(obj_Mail);       }
      // Обработчик
   void obj_Mail(object sender, string message)       {
```

событие класса GoodClass и было инициировано первым, но отработало позднее, т.к. ждало срабатывания системного таймера

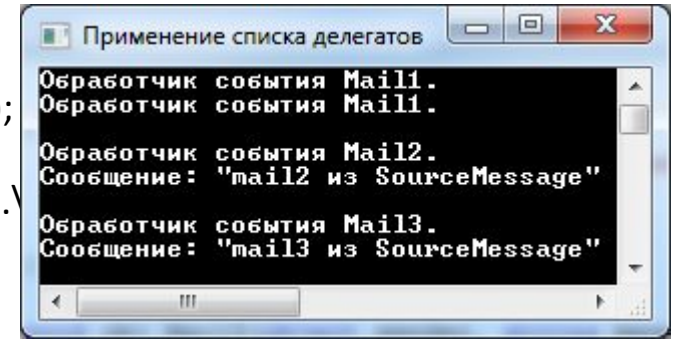


Создание событий со списком делегатов

- В библиотеке **.NET Framework** для структурированной поддержки событий создан класс **EventHandlerList**, который может хранить в себе элементы, выполняющие функцию делегатов с присоединенными к ним обработчиками. Эти элементы маркируются ключами типа **Object**. Каждому ключу соответствует один или несколько одноадресных делегатов, способных хранить вызовы обработчиков одного и того же прототипа.
- За событием закрепляется определенный ключ и при манипуляции с событием во внешнем коде делегаты, маркированные этим ключом, добавляются в список. Чтобы активизировать событие, достаточно обратиться к списку с закрепленным за событием ключом, все одноадресные делегаты будут извлечены из списка и присоединенные обработчики будут выполнены.
- Поскольку элементы списка реализуют вызов обработчиков одноадресным способом, то использование списка позволяет прикреплять обработчики не только с пустым возвращаемым значением, но и с возвращаемым значением любого типа. Хотя в большинстве случаев это несущественное преимущество, поскольку результат обработки можно вернуть и через аргументы обработчика.
  Класс **EventHandlerList** находится в пространстве имен System. ComponentModel.

```
// демонстрируется создание событий на основе списка одноадресных делегатов.
  // Класс-источник сообщения
  class SourceMessage   {
    // Создание списка делегатов вместо базовых полей
    System.ComponentModel.EventHandlerList eventList
      = new System.ComponentModel.EventHandlerList();
    // Объявление типов делегатов
    public delegate void Message1();
    public delegate int Message2(string message);
    // Непустое возвращаемое значение
    public delegate void Message3(object sender, string message);
    // Создание ключей для делегатов
    Object key1 = new Object();
    Object key2 = new Object();
    Object key3 = new Object();
    // Создание события на базе списка
    public event Message1 Mail1      {
      add        {
        eventList.AddHandler(key1, value);   // Дополняем список делегатов
      }
      remove        {
        eventList.RemoveHandler(key1, value);  // Удаляем из списка
      }     }
    // Создание события на базе списка
    public event Message2 Mail2      {
      add        {
     eventList.AddHandler(key2, value);  // Расширяем список делегатов
      }
      remove        {
        eventList.RemoveHandler(key2, value);// Удаляем из списка
      }     }
    // Создание события на базе списка
    public event Message3 Mail3      {
      add        {
        eventList.AddHandler(key3, value);   // Расширяем список делегатов
      }
```

```
   // Симуляция срабатывания события Mail3
   public void DispatchMail3()       {
      // Извекаем из списка все делегаты для Mail3, помеченные ключом
      Message3 mail3 = (Message3)eventList[key3];
      if (mail3 != null)
         mail3(this, "\"mail3 из SourceMessage\"");       }    }
// Получатель сообщения
class MyClass    {
   // Конструктор
   public MyClass()       {
      // Создаем объект с событиями
      SourceMessage obj = new SourceMessage();
      // Подписываемся на обработчики
      obj.Mail1 += new SourceMessage.Message1(obj_Mail1);
      obj.Mail1 += new SourceMessage.Message1(obj_Mail1);
      obj.Mail2 += new SourceMessage.Message2(obj_Mail2);
      obj.Mail3 += new SourceMessage.Message3(obj_Mail3);
      // Запускаем события
      obj.DispatchMail1();
      obj.DispatchMail2();
      obj.DispatchMail3();       }
   // Обработчики
   void obj_Mail1()       {
      Console.WriteLine("Обработчик события Mail1.");
   int obj_Mail2(string message)       {
      Console.WriteLine("\nОбработчик события Mail2.\
         "Сообщение: {0}\n", message);
      return 1;       }
   void obj_Mail3(object sender, string message)       {
      Console.WriteLine("Обработчик события Mail3.\n" +
```



Стандартный делегат EventHandler и стандартный аргумент EventArgs

- Для большинства практических случаев обработчикам событий достаточно двух параметров, первый из которых будет адресовать объект, возбудивший событие, а второй - некоторую дополнительную информацию.
- Для этих целей существует библиотечный делегат **EventHandler**.
- Второй аргумент делегата всегда можно расширить, если мы хотим передать с событием свою специфическую информацию.

```
// Расширение библиотечного класса аргументов
   class MyEventArgs : EventArgs    {
      private string message; // Поле для хранения сообщения
        // Сервис доступа к полю
      public String Message       {
         get { return message; }
         set { message = value; }       }    }
      // Класс-источник сообщения
   class SourceEvent    {
      // Тип стандартного делегата объявлен в mscorlib.System так
      // public delegate void EventHandler(object sender, System.EventArgs e)
       // Создание события на базе стандартного делегата
      public event EventHandler Event;
      // Метод диспетчеризации события
      protected virtual void OnEvent(EventArgs args)       {
         if (Event != null)
            Event(this, args); // Вызываем обработчики
      }
      // Симулятор срабатывания события Event по внешней причине
      public void SimulateEvent()       {
         // Создаем толстый объект и формируем передаваемую информацию
         MyEventArgs args = new MyEventArgs();
         args.Message = "'Это сообщение поступило с событием'";
         // Вызываем функцию диспетчеризации события.
         // Функция ожидает тонкий объект, а ей  передается толстый объект - это нормально
         OnEvent(args);       }    }
```

```
class MyClass    {// Получатель сообщения
      // Конструктор
 public MyClass()       {
         // Создаем объект, имеющий событие
         SourceEvent obj = new SourceEvent();
         // Подписываемся на обработчики события
         obj.Event += new EventHandler(Handler1);
         obj.Event += new EventHandler(Handler2);
         // Вызываем симулятор возникновения события
         obj.SimulateEvent();       }
      void Handler1(object sender, EventArgs e)       {
         // Хотим извлечь информацию из толстого объекта,
         // значит нужно повысить полномочия тонкой ссылки
         MyEventArgs args = (MyEventArgs)e;
         String message = args.Message;
         Console.WriteLine("(Handler1) Получена информация:\n" + message);
         Console.WriteLine();       }
         void Handler2(object sender, EventArgs e)       {
         Console.WriteLine("(Handler2) Событие из объекта {0}\n" +    "Передан объект-аргумент
      {1}",    sender.GetType().Name,      e.GetType().Name);
      }   }
    // Запуск
   class Program    {
      static void Main()       {
         // Настройка консоли
         Console.Title = "Применение делегата EventHandl
        new MyClass();    // Исполняем
}       }
```