Еще одним важным средством С#, основывающимся на делегатах, является *событие.*

Событие, по существу, представляет собой автоматическое уведомление о том,

что произошло некоторое действие. События действуют по следующему принципу:

объект, проявляющий интерес к событию, регистрирует обработчик этого события.

Когда же событие происходит, вызываются все зарегистрированные обработчики этого

события. Обработчики событий обычно представлены делегатами.

# События

События являются членами класса и объявляются с помощью ключевого слова

event. Чаще всего для этой цели используется следующая форма:

event делегат\_события имя\_события;

где *делегат\_события* обозначает имя делегата, используемого для поддержки события,

а *имя\_события* — конкретный объект объявляемого события.

(***glava15\_10***)

//delegate of event

delegate void MyEventHandler();

//class with event

class MyEvent

{

public event MyEventHandler SomeEvent;

//this method for event run

public void OnSomeEvent()

{

if (SomeEvent != null)

SomeEvent();

}

}

class EventDemo

{

//event handler

static void Handler()

{

Console.WriteLine("Event has happened;");

}

static void Main()

{

MyEvent evt = new MyEvent();

//add Handler() in event list

evt.SomeEvent += Handler;

//run event

evt.OnSomeEvent();

}

}

Несмотря на всю свою простоту, данный пример кода содержит все основные элементы,

необходимые для обработки событий. Он начинается с объявления типа делегата

для обработчика событий, как показано ниже.

delegate void MyEventHandler();

Все события активизируются с помощью делегатов. Поэтому тип делегата события

определяет возвращаемый тип и сигнатуру для события. В данном случае параметры

события отсутствуют, но их разрешается указывать.

Далее создается класс события MyEvent. В этом классе объявляется событие

SomeEvent в следующей строке кода.

public event MyEventHandler SomeEvent;

Обратите внимание на синтаксис этого объявления. Ключевое слово event уведомляет

компилятор о том, что объявляется событие.

Кроме того, в классе MyEvent объявляется метод OnSomeEvent(), вызываемый для

сигнализации о запуске события. Это означает, что он вызывается, когда происходит

событие. В методе OnSomeEvent() вызывается обработчик событий с помощью делегата

SomeEvent.

if (SomeEvent != null)

SomeEvent();

Как видите, обработчик вызывается лишь в том случае, если событие SomeEvent не

является пустым. А поскольку интерес к событию должен быть зарегистрирован в других

частях программы, чтобы получать уведомления о нем, то метод OnSomeEvent()

может быть вызван до регистрации любого обработчика события. Но во избежание

вызова по пустой ссылке делегат события должен быть проверен, чтобы убедиться в

том, что он не является пустым.

В классе EventDemo создается обработчик событий Handler(). В данном простом

примере обработчик событий просто выводит сообщение, но другие обработчики

могут выполнять более содержательные функции. Далее в методе Main() создается

объект класса события MyEvent, a Handler() регистрируется как обработчик этого

события, добавляемый в список.

MyEvent evt = new MyEvent();

//add Handler() in event list

evt.SomeEvent += Handler;

Обратите внимание на то, что обработчик добавляется в список с помощью оператора

+=. События поддерживают ***только*** операторы += и -=. В данном случае метод

Handler() является статическим, но в качестве обработчиков событий могут также

служить методы экземпляра.

И наконец, событие запускается, как показано ниже.

evt.OnSomeEvent();

# Пример групповой адресации события

Как и делегаты, события поддерживают групповую адресацию. Это дает возможность

нескольким объектам реагировать на уведомление о событии. Ниже приведен

пример групповой адресации события.

(***glava15\_11***)

//delegate for event

delegate void MyEventHandler();

//delegate with event

class MyEvent

{

public event MyEventHandler SomeEvent;

//this method call event

public void OnSomeEvent()

{

if (SomeEvent != null)

SomeEvent();

}

}

class X

{

public void Xhandler()

{

Console.WriteLine("Event got with X class object.");

}

}

class Y

{

public void Yhandler()

{

Console.WriteLine("Event got with Y class object.");

}

}

class EventDemo2

{

static void Handler()

{

Console.WriteLine("Event got with EventDemo class object.");

}

static void Main()

{

MyEvent evt = new MyEvent();

X xOb = new X();

Y yOb = new Y();

//add handler int event list

evt.SomeEvent += Handler;

evt.SomeEvent += xOb.Xhandler;

evt.SomeEvent += yOb.Yhandler;

//run event

evt.OnSomeEvent();

Console.WriteLine();

//remove handler

evt.SomeEvent -= xOb.Xhandler;

evt.OnSomeEvent();

}

}

В данном примере создаются два дополнительных класса, X и Y, в которых также

определяются обработчики событий, совместимые с делегатом MyEventHandler.

Поэтому эти обработчики могут быть также включены в цепочку событий. Обратите

внимание на то, что обработчики в классах X и Y не являются статическими. Это означает,

что сначала должны быть созданы объекты каждого из этих классов, а затем в

цепочку событий должны быть введены обработчики, связанные с их экземплярами.

Об отличиях между обработчиками экземпляра и статическими обработчиками речь

пойдет в следующем разделе.

# Методы экземпляра в сравнении со статическими

# методами в качестве обработчиков событий

Методы экземпляра и статические методы могут быть использованы в качестве

обработчиков событий, но между ними имеется одно существенное отличие. Когда

статический метод используется в качестве обработчика, уведомление о событии распространяется

на весь класс. А когда в качестве обработчика используется метод экземпляра,

то события адресуются конкретным экземплярам объектов. Следовательно,

каждый объект определенного класса, которому требуется получить уведомление о событии,

должен быть зарегистрирован отдельно. На практике большинство обработчиков

событий представляет собой методы экземпляра, хотя это, конечно, зависит от

конкретного приложения. Рассмотрим применение каждой из этих двух разновидностей

методов в качестве обработчиков событий на конкретных примерах.

В приведенной ниже программе создается класс X, в котором метод экземпляра

определяется в качестве обработчика событий. Это означает, что каждый объект класса

X должен быть зарегистрирован отдельно, чтобы получать уведомления о событиях.

Для демонстрации этого факта в данной программе производится групповая адресация

события трем отдельным объектам класса X.

(***glava15\_12***)

//delegate for event

delegate void MyEventHandler();

//class with event handler

class MyEvent

{

public event MyEventHandler SomeEvent;

//to run event

public void onSomeEvent()

{

if (SomeEvent != null)

SomeEvent();

}

}

class X

{

int id;

public X(int x) { id = x; }

//for event handler

public void Xhandler()

{

Console.WriteLine("Event handle with object: " + id);

}

}

class EventDemo3

{

static void Main()

{

MyEvent evt = new MyEvent();

X o1 = new X(1);

X o2 = new X(2);

X o3 = new X(3);

evt.SomeEvent += o1.Xhandler;

evt.SomeEvent += o2.Xhandler;

evt.SomeEvent += o3.Xhandler;

//run event

evt.onSomeEvent();

}

}

Как следует из результата выполнения кода из приведенного выше примера, каждый

объект должен зарегистрировать свой интерес в событии отдельно, и тогда он будет

получать отдельное уведомление о событии.

С другой стороны, когда в качестве обработчика событий используется статический

метод, события обрабатываются независимо от какого-либо объекта, как демонстрируется

в приведенном ниже примере программы.

(***glava15\_12.1***)

//delegate for event

delegate void MyEventHandler();

//class with event handler

class MyEvent

{

public event MyEventHandler SomeEvent;

//to run event

public void onSomeEvent()

{

if (SomeEvent != null)

SomeEvent();

}

}

class X

{

//this static method for event handler

public static void Xhandler()

{

Console.WriteLine("Event handle with class.");

}

}

class EventDemo3

{

static void Main()

{

MyEvent evt = new MyEvent();

evt.SomeEvent += X.Xhandler;

//run event

evt.onSomeEvent();

}

}

Обратите в данном примере внимание на то, что объекты класса X вообще не

создаются. Но поскольку Xhandler() является статическим методом класса X, то

он может быть привязан к событию SomeEvent и выполнен при вызове метода

OnSomeEvent().

# **Применение аксессоров событий add, remove**

В приведенных выше примерах события формировались в форме, допускавшей автоматическое

управление списком вызовов обработчиков событий, включая добавление

и удаление обработчиков событий из списка. Поэтому управление этим списком

не нужно было организовывать вручную. Благодаря именно этому свойству такие события

используются чаще всего. Тем не менее организовать управление списком вызовов

обработчиков событий можно и вручную, чтобы, например, реализовать специальный

механизм сохранения событий.

Для управления списком обработчиков событий служит расширенная форма оператора

event, позволяющая использовать *аксессоры событий.* Эти аксессоры предоставляют

средства для управления реализацией подобного списка в приведенной

ниже форме.

event делегат\_события имя\_события

{

add

{

// Код добавления события в цепочку событий.

}

remove

{

// Код удаления события из цепочки событий.

}

}

В эту форму входят два аксессора событий: add и remove. Аксессор add вызывается,

когда обработчик событий добавляется в цепочку событий с помощью оператора +=.

В то же время аксессор remove вызывается, когда обработчик событий удаляется из

цепочки событий с помощью оператора -=.

Когда вызывается аксессор add или remove, он принимает в качестве параметра

добавляемый или удаляемый обработчик. Как и в других разновидностях аксессоров,

этот неявный параметр называется value. Реализовав аксессоры add или remove,

можно организовать специальную схему хранения обработчиков событий. Например,

обработчики событий можно хранить в массиве, стеке или очереди.

Ниже приведен пример программы, демонстрирующей аксессорную форму события.

В ней для хранения обработчиков событий используется массив. Этот массив

состоит всего из трех элементов, поэтому в цепочке событий можно хранить одновременно

только три обработчика.

(***glava15\_13***)

//delegate for event

delegate void MyEventHandler();

//class with event handler

class MyEvent

{

//define class for maximum 3 events

MyEventHandler[] evnt = new MyEventHandler[3];

public event MyEventHandler SomeEvent

{

//add event in list

add

{

int i;

for (i = 0; i < 3; i++)

if (evnt[i] == null)

{

evnt[i] = value;

break;

}

if (i == 3) Console.WriteLine("Event list is full.");

}

//remove events from list

remove

{

int i;

for (i = 0; i < 3; i++)

if (evnt[i] == value)

{

evnt[i] = null;

break;

}

if (i == 3) Console.WriteLine("Event handler has't found.");

}

}

//to run event

public void onSomeEvent()

{

for (int i = 0; i < 3; i++)

if (evnt[i] != null) evnt[i](); //!!!!

}

}

class W

{

//this method for event handler

public void Whandler()

{

Console.WriteLine("Event handle with object W class.");

}

}

class X

{

//this method for event handler

public void Xhandler()

{

Console.WriteLine("Event handle with object X class.");

}

}

class Y

{

//this method for event handler

public void Yhandler()

{

Console.WriteLine("Event handle with object Y class.");

}

}

class Z

{

//this method for event handler

public void Zhandler()

{

Console.WriteLine("Event handle with object Z class.");

}

}

class EventDemo5

{

static void Main()

{

MyEvent evt = new MyEvent();

W wOb = new W();

X xOb = new X();

Y yOb = new Y();

Z zOb = new Z();

//add handlers in event list

evt.SomeEvent += wOb.Whandler;

evt.SomeEvent += xOb.Xhandler;

evt.SomeEvent += yOb.Yhandler;

//cant save list is full

evt.SomeEvent += zOb.Zhandler;

Console.WriteLine();

//run events

evt.onSomeEvent();

Console.WriteLine();

//remove handler

Console.WriteLine("Removing handler xOb.Xhandler.");

evt.SomeEvent -= xOb.Xhandler;

evt.onSomeEvent();

Console.WriteLine();

//trying delete handler one more time

Console.WriteLine("Removing handler xOb.Xhandler one more time.");

evt.SomeEvent -= xOb.Xhandler;

evt.onSomeEvent();

Console.WriteLine();

//Now add Zhandler

Console.WriteLine("Add Zhandler.");

evt.SomeEvent += zOb.Zhandler;

evt.onSomeEvent();

}

}

Рассмотрим данную программу более подробно. Сначала в ней определяется делегат

обработчиков событий MyEventHandler. Затем объявляется класс MyEvent. В самом

его начале определяется массив обработчиков событий evnt, состоящий из трех

элементов.

MyEventHandler[] evnt = new MyEventHandler[3];

Этот массив служит для хранения обработчиков событий, добавляемых в цепочку

событий. По умолчанию элементы массива evnt инициализируются пустым значением

(null).

Далее объявляется событие SomeEvent. В этом объявлении используется приведенная

ниже аксессорная форма оператора event.

public event MyEventHandler SomeEvent

{

//add event in list

add

{

int i;

for (i = 0; i < 3; i++)

if (evnt[i] == null)

{

evnt[i] = value;

break;

}

if (i == 3) Console.WriteLine("Event list is full.");

}

//remove events from list

remove

{

int i;

for (i = 0; i < 3; i++)

if (evnt[i] == value)

{

evnt[i] = null;

break;

}

if (i == 3) Console.WriteLine("Event handler has't found.");

}

}

Когда в цепочку событий добавляется обработчик событий, вызывается аксессор

add, и в первом неиспользуемом (т.е. пустом) элементе массива evnt запоминается

ссылка на этот обработчик, содержащаяся в неявно задаваемом параметре value. Если

в массиве отсутствуют свободные элементы, то выдается сообщение об ошибке. (Разумеется,

в реальном коде при переполнении списка лучше сгенерировать соответствующее

исключение.) Массив evnt состоит всего из трех элементов, поэтому в нем можно

сохранить только три обработчика событий. Когда же обработчик событий удаляется

из цепочки событий, то вызывается аксессор remove и в массиве evnt осуществляется

поиск ссылки на этот обработчик, передаваемой в качестве параметра value. Если

ссылка найдена, то соответствующему элементу массива присваивается пустое значение

(null), а значит, обработчик удаляется из цепочки событий.

При запуске события вызывается метод OnSomeEvent(). В этом методе происходит

циклическое обращение к элементам массива evnt для вызова по очереди каждого обработчика

событий.

Как демонстрирует рассматриваемый здесь пример программы, механизм хранения

обработчиков событий нетрудно реализовать, если в этом есть потребность. Но для большинства

приложений более подходящим оказывается используемый по умолчанию механизм

хранения обработчиков событий, который обеспечивает форма оператора event

без аксессоров. Тем не менее аксессорная форма оператора event используется в особых

случаях. Так, если обработчики событий необходимо выполнять в программе в порядке

их приоритетности, а не в том порядке, в каком они вводятся в цепочку событий, то для их

хранения можно воспользоваться очередью по приоритету.

# Разнообразные возможности событий

События могут быть определены и в интерфейсах. При этом события должны

предоставляться классами, реализующими интерфейсы. События могут быть также

определены как абстрактные (abstract). В этом случае конкретное событие должно

быть реализовано в производном классе. Но аксессорные формы событий не могут

быть абстрактными. Кроме того, событие может быть определено как герметичное

(sealed). И наконец, событие может быть виртуальным, т.е. его можно переопределить

в производном классе.

# **Применение анонимных методов и лямбда-**

# **выражений вместе с событиями**

Анонимные методы и лямбда-выражения особенно удобны для работы с событиями,

поскольку обработчик событий зачастую вызывается только в коде, реализующем

механизм обработки событий. Это означает, что создавать автономный метод, как правило,

нет никаких причин. А с помощью лямбда-выражений или анонимных методов

можно существенно упростить код обработки событий.

Как упоминалось выше, лямбда-выражениям теперь отдается большее предпочтение

по сравнению с анонимными методами, поэтому начнем именно с них. Ниже

приведен пример программы, в которой лямбда-выражение используется в качестве

обработчика событий.

(***glava15\_14***)

//delegate for event

delegate void MyEventHandler(int n);

//class with event

class MyEvent

{

public event MyEventHandler SomeEvent;

//this method for event run

public void OnSomeVent(int n)

{

if (SomeEvent != null)

SomeEvent(n);

}

}

class LambdaEventDemo

{

static void Main()

{

MyEvent evt = new MyEvent();

//Use lambda as event handler

evt.SomeEvent += (n) =>

Console.WriteLine("Event recieved. Value is: " + n);

//run event

evt.OnSomeVent(1);

evt.OnSomeVent(2);

}}

Обратите особое внимание на то, как в этой программе лямбда-выражение используется

в качестве обработчика событий.

//Use lambda as event handler

evt.SomeEvent += (n) =>

Console.WriteLine("Event recieved. Value is: " + n);

Синтаксис для использования лямбда-выражения в качестве обработчика событий

остается таким же, как для его применения вместе с любым другим типом делегата.

Несмотря на то что при создании анонимной функции предпочтение следует

теперь отдавать лямбда-выражениям, в качестве обработчика событий можно по-

прежнему использовать анонимный метод. Ниже приведен вариант обработчика событий

из предыдущего примера, измененный с целью продемонстрировать применение

анонимного метода.

// Использовать анонимный метод в качестве обработчика событий.

evt.SomeEvent += delegate (int n)

{

Console.WriteLine("Событие получено. Значение равно " + n);

};

Как видите, синтаксис использования анонимного метода в качестве обработчика

событий остается таким же, как и для его применения вместе с любым другим типом

делегата.

# Рекомендации по обработке событий в среде .NET Framework

В C# разрешается формировать какие угодно разновидности событий. Но ради совместимости

программных компонентов со средой .NET Framework следует придерживаться

рекомендаций, установленных для этой цели корпорацией Microsoft. Эти

рекомендации, по существу, сводятся к следующему требованию: у обработчиков событий

должны быть два параметра. Первый из них — ссылка на объект, формирующий

событие, второй — параметр типа EventArgs, содержащий любую дополнительную

информацию о событии, которая требуется обработчику. Таким образом, .NET-

совместимые обработчики событий должны иметь следующую общую форму.

void обработчик(object отправитель, EventArgs е)

{

// ...

}

Как правило, *отправитель* — это параметр, передаваемый вызывающим кодом с

помощью ключевого слова this. А параметр е типа EventArgs содержит дополнительную

информацию о событии и может быть проигнорирован, если он не нужен.

Сам класс EventArgs не содержит поля, которые могут быть использованы для

передачи дополнительных данных обработчику. Напротив, EventArgs служит в качестве

базового класса, от которого получается производньгй класс, содержащий все

необходимые поля. Тем не менее в классе EventArgs имеется одно поле Empty типа

static, которое представляет собой объект типа EventArgs без данных.

Ниже приведен пример программы, в которой формируется .NET-совместимое

событие.

(***glava15\_15***)

//define class derived from EventArgs

class MyEventArgs :EventArgs

{

public int EventNum;

}

//define delegate event type

delegate void MyEventHandler(object source, MyEventArgs arg);

//class with event

class MyEvent

{

static int count = 0;

public event MyEventHandler SomeEvent;

//this method runs event SomeEvent

public void OnSomeEvent()

{

MyEventArgs arg = new MyEventArgs();

if(SomeEvent != null)

{

arg.EventNum = count++;

SomeEvent(this, arg);

}

}

}

class X

{

public void Handler(object source, MyEventArgs arg)

{

Console.WriteLine("Event " + arg.EventNum

+ " got by X class");

Console.WriteLine("Source: " + source);

Console.WriteLine();

}

}

class Y

{

public void Handler(object source, MyEventArgs arg)

{

Console.WriteLine("Event " + arg.EventNum

+ " got by Y class");

Console.WriteLine("Source: " + source);

Console.WriteLine();

}

}

class EventDemo6

{

static void Main()

{

X ob1 = new X();

Y ob2 = new Y();

MyEvent evt = new MyEvent();

//add Handler() in event list

evt.SomeEvent += ob1.Handler;

evt.SomeEvent += ob2.Handler;

//run events

evt.OnSomeEvent();

evt.OnSomeEvent();

}

}

В данном примере создается класс MyEventArgs, производный от класса EventArgs.

В классе MyEventArgs добавляется лишь одно его собственное поле: EventNum. Затем

объявляется делегат MyEventHandler, принимающий два параметра, требующиеся

для среды .NET Framework. Как пояснялось выше, первый параметр содержит ссылку

на объект, формирующий событие, а второй параметр — ссылку на объект класса

EventArgs или производного от него класса. Обработчики событий Handler(), определяемые

в классах X и Y, принимают параметры тех же самых типов.

В классе MyEvent объявляется событие SomeEvent типа MyEventHandler. Это

событие запускается в методе OnSomeEvent() с помощью делегата SomeEvent, которому

в качестве первого аргумента передается ссылка this, а вторым аргументом

служит экземпляр объекта типа MyEventArgs. Таким образом, делегату типа

MyEventHandler передаются надлежащие аргументы в соответствии с требованиями

совместимости со средой .NET.

# Применение делегатов EventHandler<TEventArgs>

# и EventHandler

В приведенном выше примере программы объявлялся собственный делегат события.

Но как правило, в этом не никакой необходимости, поскольку в среде .NET

Framework предоставляется встроенный обобщенный делегат под названием

EventHandler<TEventArgs>. В данном случае тип TEventArgs обозначает тип аргумента, передаваемого параметру EventArgs события. Например, в приведенной выше программе событие SomeEvent может быть объявлено в классе MyEvent следующим образом.

public event EventHandler<MyEventArgs> SomeEvent;

В общем, рекомендуется пользоваться именно таким способом, а не определять

собственный делегат.

Для обработки многих событий параметр типа EventArgs оказывается ненужным.

Поэтому с целью упростить создание кода в подобных ситуациях в среду .NET

Framework внедрен необобщенный делегат типа EventHandler. Он может быть использован

для объявления обработчиков событий, которым не требуется дополнительная

информация о событиях. Ниже приведен пример использования делегата EventHandler.

(***glava15\_16***)

//class with event

class MyEvent

{

public event EventHandler SomeEvent; //using delegate EventHandler

//this method runs event

public void OnSomeEvent()

{

if (SomeEvent != null)

SomeEvent(this, EventArgs.Empty);

}

}

class EventDemo7

{

static void Handler(object source, EventArgs arg)

{

Console.WriteLine("Event has happened.");

Console.WriteLine("Source: " + source);

}

static void Main()

{

MyEvent evt = new MyEvent();

//add Handler() in event list

evt.SomeEvent += Handler;

//run event

evt.OnSomeEvent();

}

}

В данном примере параметр типа EventArgs не используется, поэтому в качестве

этого параметра передается объект-заполнитель EventArgs.Empty. Результат выполнения

кода из данного примера следующий.

# Практический пример обработки событий

События нередко применяются в таких ориентированных на обмен сообщениями

средах, как Windows. В подобной среде программа просто ожидает до тех пор, пока

не будет получено конкретное сообщение, а затем она предпринимает соответствующее

действие. Такая архитектура вполне пригодна для обработки событий средствами

С#, поскольку дает возможность создавать обработчики событий для реагирования

на различные сообщения и затем просто вызывать обработчик при получении конкретного

сообщения. Так, щелчок левой кнопкой мыши может быть связан с событием

LButtonClick. При получении сообщения о щелчке левой кнопкой мыши вызывается

метод OnLButtonClick(), и об этом событии уведомляются все зарегистрированные

обработчики.

Разработка программ для Windows, демонстрирующих такой подход, выходит

за рамки этой главы, тем не менее, рассмотрим пример, дающий представление о

принципе, по которому действует данный подход. В приведенной ниже программе

создается обработчик событий, связанных с нажатием клавиш. Всякий раз, когда

на клавиатуре нажимается клавиша, запускается событие KeyPress при вызове метода

OnKeyPress(). Следует заметить, что в этой программе формируются .NET-

совместимые события и что их обработчики предоставляются в лямбда-выражениях.

(***glava15\_17***)

//create class derived from EventArgs

//and keeps pressed keys

class KeyEventArgs : EventArgs

{

public char ch;

}

//define event class, connected with presing keys on keyboard

class KeyEvent

{

public event EventHandler<KeyEventArgs> KeyPress;

//this method calls if keys is pressed

public void OnKeyPress(char key)

{

KeyEventArgs k = new KeyEventArgs();

if (KeyPress != null)

{

k.ch = key;

KeyPress(this, k);

}

}

}

class KeyEventDemo

{

static void Main()

{

KeyEvent kevt = new KeyEvent();

ConsoleKeyInfo key;

int count = 0;

//using lambda expr to show fact of pressing

kevt.KeyPress += (sender, e) =>

Console.WriteLine("Got key pressed: " + e.ch);

//use lambda to count pressed keys

kevt.KeyPress += (sender, e) =>

count++;

Console.WriteLine("Type few symbols. Dot for the end.");

do

{

key = Console.ReadKey();

kevt.OnKeyPress(key.KeyChar);

} while (key.KeyChar != '.');

Console.WriteLine("Pressed " + count + " keys.");

}

}

В самом начале этой программы объявляется класс KeyEventArgs, производный

от класса EventArgs и служащий для передачи сообщения о нажатии клавиши обработчику

событий. Затем объявляется обобщенный делегат EventHandler, определяющий

обработчик событий, связанных с нажатием клавиш. Эти события инкапсулируются

в классе KeyEvent, где определяется событие KeyPress.

В методе Main() сначала создается объект kevt класса KeyEvent. Затем в цепочку

событий kevt.KeyPress добавляется обработчик, предоставляемый лямбда-

выражением. В этом обработчике отображается факт каждого нажатия клавиши, как

показано ниже.

kevt.KeyPress += (sender, e) =>

Console.WriteLine("Got key pressed: " + e.ch);

Далее в цепочку событий kevt.KeyPress добавляется еще один обработчик, предоставляемый

лямбда-выражением. В этом обработчике подсчитывается количество

нажатых клавиш, как показано ниже.

kevt.KeyPress += (sender, e) =>

count++;// count — это внешняя переменная

Обратите внимание на то, что count является локальной переменной, объявленной

в методе Main() и инициализированной нулевым значением.

Далее начинает выполняться цикл, в котором метод kevt.OnKeyPress() вызывается

при нажатии клавиши. Об этом событии уведомляются все зарегистрированные

обработчики событий. По окончании цикла отображается количество нажатых клавиш.

Несмотря на всю свою простоту, данный пример наглядно демонстрирует саму

суть обработки событий средствами С#. Аналогичный подход может быть использован

и для обработки других событий. Безусловно, в некоторых случаях анонимные обработчики

событий могут оказаться непригодными, и тогда придется внедрить именованные

методы.