*Делегат* предоставляет возможность инкапсулировать метод,а *событие* уведомляет о том, что произошло некотороедействие. Делегаты и события тесно связаны друг с другом,поскольку событие основывается на делегате. Оба средстварасширяют круг прикладных задача, решаемых при программированиина С#.

# Делегаты

Начнем с определения понятия делегата. Попросту говоря, *делегат* представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается. Ниже будет показано, насколько действенным оказывается такой принцип.

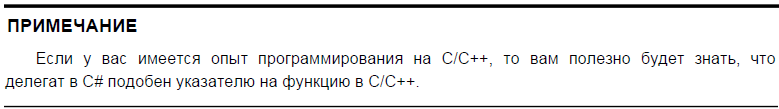
Следует особо подчеркнуть, что один и тот же делегат может быть использован

для вызова разных методов во время выполнения программы, для чего достаточно

изменить метод, на который ссылается делегат. Таким образом, метод, вызываемый

делегатом, определяется во время выполнения, а не в процессе компиляции. В этом,

собственно, и заключается главное преимущество делегата.



Тип делегата объявляется с помощью ключевого слова delegate. Ниже приведена общая форма объявления делегата:

delegate возвращаемый\_тип имя(список\_параметров);

где *возвращаемый\_тип* обозначает тип значения, возвращаемого методами, которые

будут вызываться делегатом; *имя* — конкретное имя делегата; *список\_параметров —*

параметры, необходимые для методов, вызываемых делегатом. Как только будет создан

экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый

тип и параметры которых соответствуют указанным в объявлении делегата.

Самое главное, что делегат может служить для вызова *любого* метода с соответствующей

сигнатурой и возвращаемым типом. Более того, вызываемый метод может быть

методом экземпляра, связанным с отдельным объектом, или же статическим методом,

связанным с конкретным классом. Значение имеет лишь одно: возвращаемый тип

и сигнатура метода должны быть согласованы с теми, которые указаны в объявлении

делегата.

(***glava15\_1***)

delegate string StrMod(string str);

class DelegateTest

{

//change spaces to -

static string ReplaceSpaces(string s)

{

Console.WriteLine("Replacing spaces.");

return s.Replace(' ', '-');

}

//delete spaces

static string RemoveSpaces(string s)

{

string temp = "";

int i;

Console.WriteLine("Deleting spaces.");

for (i = 0; i < s.Length; i++)

if (s[i] != ' ') temp += s[i];

return temp;

}

//string backwards

static string Reverse(string s)

{

string temp = "";

int i, j;

Console.WriteLine("Reversing string");

for (j = 0, i = s.Length - 1; i >= 0; i--, j++)

temp += s[i];

return temp;

}

static void Main()

{

//construct delegate

StrMod strOp = new StrMod(ReplaceSpaces);

string str;

//call method with delegate

str = strOp("Just a test.");

Console.WriteLine("Result: " + str);

Console.WriteLine();

strOp = new StrMod(RemoveSpaces);

str = strOp("Just a test.");

Console.WriteLine("Result: " + str);

Console.WriteLine();

strOp = new StrMod(Reverse);

str = strOp("Just a test.");

Console.WriteLine("Result: " + str);

Console.WriteLine();

}

}

Как видите, делегат StrMod принимает один параметр типа string и возвращает

одно значение того же типа.

В методе Main() создается переменная экземпляра strOp ссылочного типа StrMod

и затем ей присваивается ссылка на метод ReplaceSpaces(). Обратите особое внимание

на следующую строку кода.

StrMod strOp = new StrMod(ReplaceSpaces);

В этой строке метод ReplaceSpaces() передается в качестве параметра. При этом

указывается только его имя, но не параметры. Данный пример можно обобщить: при

получении экземпляра делегата достаточно указать только имя метода, на который

должен ссылаться делегат. Ясно, что сигнатура метода должна совпадать с той, что

указана в объявлении делегата. В противном случае во время компиляции возникнет

ошибка.

Далее метод ReplaceSpaces() вызывается с помощью экземпляра делегата strOp,

как показано ниже.

str = strOp("Just a test.");

Экземпляр делегата strOp ссылается на метод ReplaceSpaces(), и поэтому вызывается

именно этот метод.

# Групповое преобразование делегируемых методов

Так называемое *групповое преобразование методов,* позволяющее присвоить имя метода делегату, ***не прибегая*** к оператору***new*** или явному вызову конструктора делегата.

strOp = ReplaceSpaces;//групповое преобразование методов

вместо

StrMod strOp = new StrMod(ReplaceSpaces);

В этой строке кода имя метода присваивается непосредственно экземпляру делегата

strOp, а все заботы по автоматическому преобразованию метода в тип делегата

*"*возлагаются*"* на средства С#. Этот синтаксис может быть распространен на любую

ситуацию, в которой метод присваивается или преобразуется в тип делегата. Синтаксис группового преобразования методов существенно упрощен по сравнению с прежним подходом к делегированию, поэтому в остальной части книги используется именно он.

# Применение методов экземпляра в качестве делегатов

В предыдущем примере использовались статические методы, но делегат может

ссылаться и на методы экземпляра, хотя для этого требуется ссылка на объект. Так,

ниже приведен измененный вариант предыдущего примера, в котором операции со

строками инкапсулируются в классе StringOps. Следует заметить, что в данном случае

может быть также использован синтаксис группового преобразования методов.

(***glava15\_2***)

Все тоже самое что и в прошлом листинге, крмое того что методы становятся public.

А экземпляр делегата принимает ссылку на метод через объект класса.

//define delegate

delegate string StrMod(string str);

class StringOps

{

//change spaces to -

public string ReplaceSpaces(string s)

{

Console.WriteLine("Replacing spaces.");

return s.Replace(' ', '-');

}

}

class DelegateTest

{

static void Main()

{

StringOps so = new StringOps(); //object of class

//initialize delegate

StrMod strOp = so.ReplaceSpaces;}

Результат выполнения этого кода получается таким же, как и в предыдущем примере,

но на этот раз делегат обращается к методам по ссылке на экземпляр объекта

класса StringOps.

# Групповая адресация

Одним из самых примечательных свойств делегата является поддержка групповой

адресации. Попросту говоря, *групповая адресация* — это возможность создать *список,*

или *цепочку вызовов,* для методов, которые вызываются автоматически при обращении

к делегату. Создать такую цепочку нетрудно. Для этого достаточно получить экземпляр

делегата, а затем добавить методы в цепочку с помощью оператора + или +=.

Для удаления метода из цепочки служит оператор - или -=. Если делегат возвращает

значение, то им становится значение, возвращаемое последним методом в списке вызовов.

***Поэтому делегат, в котором используется групповая адресация, обычно имеет***

***возвращаемый тип void***.

Ниже приведен пример групповой адресации. Это переработанный вариант

предыдущих примеров, в котором тип значений, возвращаемых методами манипулирования

строками, изменен на void, а для возврата измененной строки в вызывающую

часть кода служит параметр типа ref. Благодаря этому методы оказываются более

приспособленными для групповой адресации.

(***glava15\_3***)

//define delegate

delegate void StrMod(ref string str);

class MultiCastDemo

{

//change spaces to -

static void ReplaceSpaces(ref string s)

{

Console.WriteLine("Replacing spaces.");

s = s.Replace(' ', '-');

}

//delete spaces

static void RemoveSpaces(ref string s)

{

string temp = "";

int i;

Console.WriteLine("Deleting spaces.");

for (i = 0; i < s.Length; i++)

if (s[i] != ' ') temp += s[i];

s = temp;

}

//string backwards

static void Reverse(ref string s)

{

string temp = "";

int i, j;

Console.WriteLine("Reversing string");

for (j = 0, i = s.Length - 1; i >= 0; i--, j++)

temp += s[i];

s = temp;

}

static void Main()

{

//build delegates

StrMod strOp;

StrMod replaceSp = ReplaceSpaces;

StrMod removeSp = RemoveSpaces;

StrMod reverseStr = Reverse;

string str = "Just test.";

//group cast

strOp = replaceSp;

strOp += reverseStr;

//call delegate with multicast

strOp(ref str);

Console.WriteLine("Result: " + str);

Console.WriteLine();

//delete method replaceSp, add method RemoveSp

strOp -= replaceSp;

strOp += removeSp;

str = "Just test."; //update

//call delegate with multicast

strOp(ref str);

Console.WriteLine("Result: " + str);

Console.WriteLine();

}

}

Как мне сразу показалось, что делегат просто указывает на какуе – то из функций и выполняет его в данном пример выше. На самом же деле, он выполняет все функции, которые включил в себя.

В методе Main() из рассматриваемого здесь примера кода создаются четыре экземпляра

делегата. Первый из них, strOp, является пустым, а три остальных ссылаются на

конкретные методы видоизменения строки. Затем организуется групповая адресация

для вызова методов RemoveSpaces() и Reverse(). Это делается в приведенных ниже

строках кода.

strOp = replaceSp;

strOp += reverseStr;

Сначала делегату strOp присваивается ссылка replaceSp, а затем с помощью оператора

+= добавляется ссылка reverseStr. При обращении к делегату strOp вызываются

оба метода, заменяя пробелы дефисами и обращая строку, как и показывает

приведенный выше результат.

Далее ссылка replaceSp удаляется из цепочки вызовов в следующей строке кода:

strOp -= replaceSp;

и добавляется ссылка removeSp в строке кода.

strOp += removeSp;

После этого вновь происходит обращение к делегату strOp. На этот раз обращается

строка с удаленными пробелами.

Цепочки вызовов являются весьма эффективным механизмом, поскольку они позволяют

определить ряд методов, выполняемых единым блоком. Благодаря этому

улучшается структура некоторых видов кода. Кроме того, цепочки вызовов имеют особое

значение для обработки событий, как станет ясно в дальнейшем.

# Ковариантность и контравариантность

Делегаты становятся еще более гибкими средствами программирования благодаря

двум свойствам: *ковариантности* и *контравариантности.* Как правило, метод, передаваемый

делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат.

Но в отношении производных типов это правило оказывается не таким строгим благодаря

ковариантности и контравариантности. В частности, ***ковариантность позволяет***

***присвоить делегату метод, возвращаемым типом которого служит класс, производный***

***от класса, указываемого в возвращаемом типе делегата. А контравариантность позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата***.

Ниже приведен пример, демонстрирующий ковариантность и контравариантность.

(***glava15\_4***)

class X

{

public int Val;

}

//class Y, defived X

class Y : X { }

//this delegate returns object of X class

//and gets object of Y class as argument

delegate X ChangeIt(Y obj);

class CoContrVariance

{

//this method returns object of X class

//and has object of X class as argument

static X IncrA(X obj)

{

X temp = new X();

temp.Val = obj.Val + 1;

return temp;

}

//this method returns object of Y class

//and has object of Y class as argument

static Y IncrB(Y obj)

{

Y temp = new Y();

temp.Val = obj.Val + 1;

return temp;

}

static void Main()

{

Y Yob = new Y();

//in this case parametr of method IncrA is object of X class,

//and argument of delegate ChangeIt - object of Y class. But

//for contrvariance next string of code is valid.

ChangeIt change = IncrA;

X Xob = change(Yob);

Console.WriteLine("Xob: " + Xob.Val);

//in this case parametr of method IncrB is object of Y class,

//and argument of delegate ChangeIt - object of X class. But

//for covariance next string of code is valid.

change = IncrB;

Yob = (Y)change(Yob);

Console.WriteLine("Yob: " + Yob.Val);

}

}

В данном примере класс Y является производным от класса X. А делегат ChangeIt

объявляется следующим образом.

delegate X ChangeIt(Y obj);

Делегат возвращает объект класса X и принимает в качестве параметра объект класса

Y. А методы IncrA() и IncrB() объявляются следующим образом.

static X IncrA(X obj)

static Y IncrB(Y obj)

Метод IncrA() принимает объект класса X в качестве параметра и возвращает

объект того же класса. А метод IncrB() принимает в качестве параметра объект класса

Y и возвращает объект того же класса. Но благодаря ковариантности и контравариантности

любой из этих методов может быть передан делегату ChangeIt, что и демонстрирует

рассматриваемый здесь пример.

Таким образом, в строке

ChangeIt change = IncrA;

метод IncrA() может быть передан делегату благодаря контравариантности, так как

объект класса X служит в качестве параметра метода IncrA(), а объект класса Y —

в качестве параметра делегата ChangeIt. Но метод и делегат оказываются совместимыми

в силу контравариантности, поскольку типом параметра метода, передаваемого

делегату, служит класс, являющийся базовым для класса, указываемого в качестве типа

параметра делегата.

Приведенная ниже строка кода также является вполне допустимой, но на этот раз

благодаря ковариантности.

change = IncrB;

В данном случае возвращаемым типом для метода IncrB() служит класс Y, а для

делегата — класс X. Но поскольку возвращаемый тип метода является производным

классом от возвращаемого типа делегата, то оба оказываются совместимыми в силу

ковариантности.

Короче говоря, ковариантность и контрвариантность в делегатах означает – что делегат на метод который оперирует методами где параметром или возвращаемым методом являеются классы, которые связаны друг с другом наследованием, может использовать этот метод и не важно как в делегате были расстановлены значения в return или аргументах.

# Класс System.Delegate

Все делегаты и классы оказываются производными неявным образом от класса

System.Delegate. Как правило, членами этого класса не пользуются непосредственно,

и это не делается явным образом в данной книге. Но члены класса System.

Delegate могут оказаться полезными в ряде особых случаев.

# Назначение делегатов

В предыдущих примерах был наглядно продемонстрирован внутренний механизм

действия делегатов, но эти примеры не показывают их истинное назначение. Как правило,

делегаты применяются по двум причинам. Во-первых, как упоминалось ранее

в этой главе, делегаты поддерживают события. И во-вторых, делегаты позволяют вызывать

методы во время выполнения программы, не зная о них ничего определенного

в ходе компиляции. Это очень удобно для создания базовой конструкции, допускающей

подключение отдельных программных компонентов. Рассмотрим в качестве

примера графическую программу, аналогичную стандартной сервисной программе

Windows Paint. С помощью делегата можно предоставить пользователю возможность

подключать специальные цветные фильтры или анализаторы изображений. Кроме

того, пользователь может составлять из этих фильтров или анализаторов целые последовательности.

Подобные возможности программы нетрудно обеспечить, используя

делегаты.

# Анонимные методы

Анонимный метод — один из способов создания безымянного блока кода, связанного

с конкретным экземпляром делегата. Для создания анонимного метода достаточно

указать кодовый блок после ключевого слова delegate. Покажем, как это делается,

на конкретном примере. В приведенной ниже программе анонимный метод служит

для подсчета от 0 до 5.

(***glava15\_5***)

//define type of delegate

delegate void CountIt();

class AnonMethDemo

{

static void Main()

{

//count nums passed to delegate as anonym

CountIt count = delegate

{

//this block passed to delegate

for (int i = 0; i <= 5; i++)

Console.WriteLine(i);

};//dont forget semicolon

count();

}

}

В данной программе сначала объявляется тип делегата CountIt без параметров

и с возвращаемым типом void. Далее в методе Main() создается экземпляр count

делегата CountIt, которому передается кодовый блок, следующий после ключевого

слова delegate. Именно этот кодовый блок и является анонимным методом, который

будет выполняться при обращении к делегату count. Обратите внимание на то,

что после кодового блока следует точка с запятой, фактически завершающая оператор

объявления.

# Передача аргументов анонимному методу

Анонимному методу можно передать один или несколько аргументов. Для этого

достаточно указать в скобках список параметров после ключевого слова delegate,

а при обращении к экземпляру делегата — передать ему соответствующие аргументы.

В качестве примера ниже приведен вариант предыдущей программы, измененный

с целью передать в качестве аргумента конечное значение для подсчета.

(***glava15\_5.1***)

//now delegate has parametr

delegate void CountIt(int end);

class AnonMethDemo2

{

static void Main()

{

//pass value to anonym method

CountIt count = delegate (int end)

{

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

Console.WriteLine(i);

};//dont forget semicolon

count(5);

Console.WriteLine();

count(5);

}

}

В этом варианте программы делегат CountIt принимает целочисленный аргумент.

Обратите внимание на то, что при создании анонимного метода список параметров

указывается после ключевого слова delegate.

# Возврат значения из анонимного метода

Анонимный метод может возвращать значение. Для этой цели служит оператор

return, действующий в анонимном методе таким же образом, как и в именованном

методе. Как и следовало ожидать, тип возвращаемого значения должен быть совместим

с возвращаемым типом, указываемым в объявлении делегата. В качестве примера

ниже приведен код, выполняющий подсчет с суммированием и возвращающий

результат.

(***glava15\_5.2***)

//this delegate returns value

delegate int CountIt(int end);

class AnonMethodDemo3

{

static void Main()

{

int result;

//end value for counting passed to anonym method

//and returns sum of numbers

CountIt count = delegate (int end)

{

int sum = 0;

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

{

Console.WriteLine(i);

sum += i;

}

return sum;

};

result = count(3);

Console.WriteLine("Sum of 3: " + result);

result = count(5);

Console.WriteLine("Sum of 5: " + result);

}

}

В этом варианте кода суммарное значение возвращается кодовым блоком, связанным

с экземпляром делегата count. Обратите внимание на то, что оператор return

применяется в анонимном методе таким же образом, как и в именованном методе.

# Применение внешних переменных в анонимных методах

Локальная переменная, в область действия которой входит анонимный метод, называется

*внешней переменной.* Такие переменные доступны для использования в анонимном

методе. И в этом случае внешняя переменная считается *захваченной.* Захваченная

переменная существует до тех пор, пока захвативший ее делегат не будет собран в *"*мусор*"*. Поэтому если локальная переменная, которая обычно прекращает свое существование

после выхода из кодового блока, используется в анонимном методе, то

она продолжает существовать до тех пор, пока не будет уничтожен делегат, ссылающийся

на этот метод.

Захват локальной переменной может привести к неожиданным результатам. В качестве

примера рассмотрим еще один вариант программы подсчета с суммированием

чисел. В данном варианте объект CountIt конструируется и возвращается статическим

методом Counter(). Этот объект использует переменную sum, объявленную в охватывающей

области действия метода Counter(), а не самого анонимного метода. Поэтому

переменная sum захватывается анонимным методом. Метод Counter() вызывается

в методе Main() для получения объекта CountIt, а следовательно, переменная sum не

уничтожается до самого конца программы.

(***glava15\_6***)

//this delegate returns value and gets parameter

delegate int CountIt(int end);

class VarCapture

{

static CountIt Counter()

{

int sum = 0;

CountIt ctObj = delegate (int end)

{

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

{

Console.WriteLine(i);

sum += i;

}

return sum;

};

return ctObj;

static void Main()

{

//get result of count

CountIt count = Counter();

int result;

result = count(3);

Console.WriteLine("Sum of 3: " + result);

result = count(5);

Console.WriteLine("Sum of 5: " + result);

}

}

Как видите, подсчет по-прежнему выполняется как обычно. Но обратите внимание

на то, что сумма 5 теперь равна 21, а не 15! Дело в том, что переменная sum захватывается

объектом ctObj при его создании в методе Counter(). Это означает, что она

продолжает существовать вплоть до уничтожения делегата count при *"*сборке мусора*"*

в самом конце программы. Следовательно, ее значение не уничтожается после возврата

из метода Counter() или при каждом вызове анонимного метода, когда происходит

обращение к делегату count в методе Main().

Несмотря на то что применение захваченных переменных может привести к довольно

неожиданным результатам, как в приведенном выше примере, оно все же

логически обоснованно. Ведь когда анонимный метод захватывает переменную, она

продолжает существовать до тех пор, пока используется захватывающий ее делегат.

В противном случае захваченная переменная оказалась бы неопределенной, когда она

могла бы потребоваться делегату.