Лямбда-выражение — это другой собой создания анонимной функции. (Первый

ее способ, анонимный метод, был рассмотрен в предыдущем разделе.) Следовательно,

лямбда-выражение может быть присвоено делегату. А поскольку лямбда-выражение

считается более эффективным, чем эквивалентный ему анонимный метод то в большинстве

случаев рекомендуется отдавать предпочтение именно ему.

**Лямбда-оператор**

Во всех лямбда-выражениях применяется новый лямбда-оператор =>, который разделяет

лямбда-выражение на две части. В левой его части указывается входной параметр

(или несколько параметров), а в правой части — тело лямбда-выражения. Оператор

=> иногда описывается такими словами, как *"*переходит*"* или *"*становится*"*.

В C# поддерживаются две разновидности лямбда-выражений в зависимости от тела

самого лямбда-выражения. Так, если тело лямбда-выражения состоит из одного выражения,

то образуется *одиночное лямбда-выражение.* В этом случае тело выражения не

заключается в фигурные скобки. Если же тело лямбда-выражения состоит из блока

операторов, заключенных в фигурные скобки, то образуется *блочное лямбда-выражение.*

При этом блочное лямбда-выражение может содержать целый ряд операторов, в том

числе циклы, вызовы методов и условные операторы if. Обе разновидности лямбда-

выражений рассматриваются далее по отдельности.

**Одиночные лямбда-выражения**

В одиночном лямбда-выражении часть, находящаяся справа от оператора =>, воздействует

на параметр (или ряд параметров), указываемый слева. Возвращаемым

результатом вычисления такого выражения является результат выполнения лямбда-

оператора.

Ниже приведена общая форма одиночного лямбда-выражения, принимающего

единственный параметр.

**параметр => выражение**

Если же требуется указать несколько параметров, то используется следующая форма.

***(список\_параметров) => выражение***

Таким образом, когда требуется указать два параметра или более, их следует заключить

в скобки. Если же *выражение* не требует параметров, то следует использовать пустые скобки.

Лямбда-выражение применяется в два этапа. Сначала объявляется тип делегата, совместимый

с лямбда-выражением, а затем экземпляр делегата, которому присваивается

лямбда-выражение. После этого лямбда-выражение вычисляется при обращении к

экземпляру делегата. Результатом его вычисления становится возвращаемое значение.

В приведенном ниже примере программы демонстрируется применение двух одиночных

лямбда-выражений. Сначала в этой программе объявляются два типа делегатов.

Первый из них, Incr, принимает аргумент типа int и возвращает результат того

же типа. Второй делегат, IsEven, также принимает аргумент типа int, но возвращает

результат типа bool. Затем экземплярам этих делегатов присваиваются одиночные

лямбда-выражения. И наконец, лямбда-выражения вычисляются с помощью соответствующих

экземпляров делегатов.

(***glava15\_7***)

//define delegate, it takes argument of type int

//and returns int

delegate int Incr(int v);

//define delegate, it itakes argument type int

//and returns type bool

delegate bool IsEven(int v);

class SimpleLambdaDemo

{

static void Main()

{

//create delegate Incr, it points to lambda func

//increases its parameter on 2

Incr incr = count => count + 2;

//now use lambda incr

Console.WriteLine("Using lambda incr: ");

int x = -10;

while(x <= 0)

{

Console.Write(x + " ");

x = incr(x);//increase value x by 2

}

Console.WriteLine("\n");

//create delegate IsEven, it points on lambda func

//returns logic true, if its parametr has even value

//else - returns false

IsEven isEven = n => n % 2 == 0;

//now use lambda isEven

Console.WriteLine("Using lambda isEven: ");

for (int i = 0; i <= 10; i++)

if (isEven(i)) Console.WriteLine(i + " even.");

}

}

Обратите в данной программе особое внимание на следующие строки объявлений.

Incr incr = count => count + 2;

IsEven isEven = n => n % 2 == 0;

В первой строке объявления экземпляру делегата incr присваивается одиночное

лямбда-выражение, возвращающее результат увеличения на 2 значения параметра

count. Это выражение может быть присвоено делегату Incr, поскольку оно совместимо

с объявлением данного делегата. Аргумент, указываемый при обращении к экземпляру

делегата incr, передается параметру count, который и возвращает результат

вычисления лямбда-выражения. Во второй строке объявления делегату isEven присваивается

выражение, возвращающее логическое значение true, если передаваемый

ему аргумент оказывается четным, а иначе — логическое значение false. Следовательно,

это лямбда-выражение совместимо с объявлением делегата IsEven.

В связи со всем изложенным выше возникает резонный вопрос: каким образом

компилятору становится известно о типе данных, используемых в лямбда-

выражении, например, о типе int параметра count в лямбда-выражении, присваиваемом

экземпляру делегата incr? Ответить на этот вопрос можно так: компилятор

делает заключение о типе параметра и типе результата вычисления выражения

по типу делегата. Следовательно, параметры и возвращаемое значение лямбда-

выражения должны быть совместимы по типу с параметрами и возвращаемым значением

делегата.

Несмотря на всю полезность логического заключения о типе данных, в некоторых

случаях приходится явно указывать тип параметра лямбда-выражения. Для этого достаточно

ввести конкретное название типа данных. В качестве примера ниже приведен

другой способ объявления экземпляра делегата incr.

Incr incr = (int count) => count + 2;

Как видите, count теперь явно объявлен как параметр типа int. Обратите также

внимание на использование скобок. Теперь они необходимы. (Скобки могут быть опущены

только в том случае, если задается лишь один параметр, а его тип явно не указывается.)

В предыдущем примере в обоих лямбда-выражениях использовался единственный

параметр, но в целом у лямбда-выражений может быть любое количество параметров,

в том числе и нулевое. Если в лямбда-выражении используется несколько параметров,

их *необходимо* заключить в скобки. Ниже приведен пример использования лямбда-

выражения с целью определить, находится ли значение в заданных пределах.

(low, high, val) => val >= low && val <= high;

А вот как объявляется тип делегата, совместимого с этим лямбда-выражением.

delegate bool InRange(int lower, int upper, int v);

Следовательно, экземпляр делегата InRange может быть создан следующим образом.

InRange rangeOK = (low, high, val) => val >= low && val <= high;

И последнее замечание: внешние переменные могут использоваться и захватываться

в лямбда-выражениях таким же образом, как и в анонимных методах.

**Блочные лямбда-выражения**

Второй разновидностью является *блочное лямбда-выражение.* Для такого лямбда-выражения характерны расширенные возможности выполнения различных операций, поскольку в его теле допускается указывать несколько операторов. Например, в блочном лямбда-выражении

можно использовать циклы и условные операторы if, объявлять переменные и т.д.

Создать блочное лямбда-выражение нетрудно. Для этого достаточно заключить тело

выражения в фигурные скобки. Помимо возможности использовать несколько операторов,

в остальном блочное лямбда-выражение, практически ничем не отличается от

только что рассмотренного одиночного лямбда-выражения.

Ниже приведен пример использования блочного лямбда-выражения для вычисления

и возврата факториала целого значения.

(***glava15\_8***)

//delegate IntOp gets one argument

//and returns result

delegate int IntOp(int end);

class StatementLambdaDemo

{

static void Main()

{

//block lambda expression returns factorial

IntOp fact = n =>

{

int r = 1;

for (int i = 1; i <= n; i++)

r = i \* r;

return r;

};//semicolon!!!

Console.WriteLine("Factorial of 3 = " + fact(3));

Console.WriteLine("Factorial of 5 = " + fact(5));

}

}

В приведенном выше примере обратите внимание на то, что в теле блочного

лямбда-выражения объявляется переменная r, организуется цикл for и используется

оператор return. Все эти элементы вполне допустимы в блочном лямбда-выражении.

И в этом отношении оно очень похоже на анонимный метод. Следовательно, многие

анонимные методы могут быть преобразованы в блочные лямбда-выражения при

обновлении унаследованного кода. И еще одно замечание: когда в блочном лямбда-

выражении встречается оператор return, он просто обусловливает возврат из лямбда-

выражения, но не возврат из охватывающего метода.

И в заключение рассмотрим еще один пример, демонстрирующий блочное лямбда-

выражение в действии. Ниже приведен вариант первого примера из этой главы, измененного

с целью использовать блочные лямбда-выражения вместо автономных методов

для выполнения различных операций со строками.

(***glava15\_9***)

//delegate

delegate string StrMod(string s);

class UseStatementLambdas

{

static void Main()

{

//create delegates, points to lambda expressions

//doing some operations with strings

//replace spaces with hyphen

StrMod ReplaceSpaces = s =>

{

Console.WriteLine("Change spaces with hyphens.");

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

};

//remove spaces

StrMod RemoveSpaces = s =>

{

Console.WriteLine("Removing spaces.");

string temp = "";

int i;

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

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

return temp;

};

//reverse string

StrMod Reverse = s =>

{

Console.WriteLine("Reversing string.");

string temp = "";

int i;

for (i = s.Length - 1; i != 0; i--)

temp += s[i];

return temp;

};

string str;

//use lambda expr with delegates

StrMod strOp = ReplaceSpaces;

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

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

Console.WriteLine();

strOp = RemoveSpaces;

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

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

Console.WriteLine();

strOp = Reverse;

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

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

Console.WriteLine();

}

}