**Замыкания в языке программирования C#**

Возможно, вы никогда не слышали о таком понятии как замыкания (closure), или слышали, но все эти функциональные штучки показались вам настолько сложным, что вы решили отложить его изучение до тех славных времен, когда компиляторы будут писать программы за вас или хотя бы в компьютерах будет предусмотрен отдельный процессор для сборщика мусора. Но не зависимо от ваших знаний и представлений о замыканиях, можно сказать с уверенностью, что вы, будучи программистом C#, не раз сталкивались с этим понятием в своей повседневной деятельности (конечно, я рассчитываю, что вы не застряли в прошлом и перешли хотя бы на C# 2.0), поэтому знать, что же это такое будет не лишним.

**Основные определения**

Давайте посмотрим на некоторые популярные определения термина “замыкание”:

*Замыкание (англ. closure) — это процедура, которая ссылается на свободные переменные в своём лексическом контексте.  
Замыкание, так же как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе.*

*Замыкания (closures) представляют собой фрагменты (блоки) кода, который можно использовать в качестве аргументов функций и методов.*

В процессе понимания таких определений неподготовленный читатель вполне может вывихнуть свой мозг и потерять охоту от изучения этого понятия навсегда. Но, на самом деле, все не так страшно. Если говорить проще и не столь формально, да еще в контексте языка C#, то идея замыканий состоит *в доступе анонимного метода не только к переданным параметрам, но и к внешнему окружению* (если пока что не совсем ясно, что такое “внешнее окружение”, то это будет понятно позднее).

Прежде чем мы перейдем к примерам, нужно дать еще несколько определений, необходимых для понимания механизма замыканий в языке C#.

*Внешняя переменная**(outer variable)*– это локальная переменная или параметр (за исключением ref- и out-параметров), доступные внутри метода, в котором объявлен анонимный метод (ключевое слово *this* можно так же рассматривать как локальную переменную, поэтому внутри анонимного метода можно получить доступ ко всем членам текущего класса).

*Захваченная внешняя переменная**(captured outer variable)* или просто *захваченная переменная* *(captured variable) –*это внешняя переменная, используемая внутри анонимного метода.

**Пример замыканий в языке C#**

Чтобы было несколько проще переварить все эти термины, приведем простой пример. Предположим, вам нужно выполнить некоторую операцию асинхронно и вместо того, чтобы создавать новый поток руками или использовать асинхронный вызов делегатов вы решили воспользоваться помощью класса *ThreadPool*.

void EnclosingMethod(bool outerVariable1, //1

    ref int nonOuterVariable) //2

{

    int outerVariable2 = 10; //3

    string capturedVariable = "captured"; //4

    if (outerVariable2 % 2 == 0)

    {

        int normalLocalVariable = 5; //5

        Console.WriteLine("Normal local variable: {0}", normalLocalVariable);

    }

    WaitCallback d = delegate(object o)

    {

        int anonymousMethodLocalVariable = 12; //6

        Console.WriteLine("Captured variable is {0}", capturedVariable);

    };

    ThreadPool.QueueUserWorkItem(d, null);

}

В методе *EnclosingMethod* продемонстрированы различные типы переменных и два способа передачи переменных внутрь анонимного метода: с помощью явного параметра и с помощью захвата внешней переменной. Переменные (1) и (3) являются внешними, по отношению к анонимному методу, на который ссылается делегат *d*,поскольку они объявлены во “внешнем окружении” по отношению к этому методу. Переменная (2) не является внешней, поскольку захват ref- или out-параметров не допускается. Переменная (4) является захваченной внешней переменной, поскольку она используется внутри анонимного метода. Переменная (5) является обычной локальной переменной и не является внешней, поскольку внутри области видимости, в которой она объявлена, не существует анонимных методов. Переменная (6) является обычной локальной переменной анонимного метода.

(Теперь самое время вернуться к списку определений и прочитать их еще раз, возможно на этот раз в них окажется больше смысла).

**ПРИМЕЧАНИЕ. Используйте анонимные методы с умом**  
Большинство опытных разработчиков знает, что не нужно бросаться с руками и ногами на каждую новомодную фишку языка программирования и использовать ее где попало. Тоже самое относится и к анонимным методам. Анонимные методы – это очень полезная возможность, которая может как существенно упростить понимание кода, так и усложнить его. Не существует формальных правил, которые бы определяли, когда следует применять анонимные методы, а когда лучше создать обыкновенный именованный метод (хотя Джеффри Рихтер придерживается правила, что любой анонимный метод, длиннее 3-х строк должен быть преобразован в именованный). Основная польза анонимного метода будет в том случае, когда он является логической частью какого-то другого метода  и не имеет особого смысла без этого контекста. Если же метод является самостоятельным, полностью выполняет некоторую задачу и может быть вызываться в различных условиях, то стоит подумать о создании именованного метода.

**Внутренняя реализация замыканий**

Теперь вам должно быть более или менее понятно, о чем идет речь и пришло время рассмотреть внутреннюю реализацию замыканий компилятором.

Давайте, в качестве примера, рассмотрим один интересный момент: а что если анонимный метод будет захватывать внешнюю переменную значимого типа (value type) и время жизни анонимного метода будет превосходить время жизни переменных в стеке? Этого очень просто добиться путем возвращения делегата, ссылающегося на анонимный метод, в качестве возвращаемого значение, сохранение этого делегата в качестве поля класса, использование в качестве функции потока или асинхронного вызова… думаю вы поняли, что возможностей очень много. Давайте рассмотрим простой пример:

static Action CreateAction()

{

    int count = 0;

    Action action = () =>

    {

        count++;

        Console.WriteLine("Count = {0}", count);

    };

    return action;

}

static void Main(string[] args)

{

    var action = CreateAction();

    action();

    action();

}

Результат выполнения программы:  
Count = 1  
Count = 2

(Здесь и в дальнейшем вместо синтаксиса анонимных делегатов C# 2.0 я буду использовать более простой и лаконичный синтаксис лямбда-выражений, которые появились в C# 3.0)

С первого взгляда может показаться, что ничего хорошего при выполнении этого кода мы не получим, поскольку переменная *count* располагается в стеке метода *CreateAction*, а вызов делегата будет осуществляться после завершения выполнения этого метода. Но дело все в том, что в стеке метода *CreateAction*нет никакой переменной *count.* Вместо этого компилятор создает анонимный класс, с открытым полем и создает метод, тело которого соответствует телу объявленного пользователем анонимного метода. Давайте более детально посмотрим, во что компилятор преобразует этот код.

static Action CreateAction()

{

    // Никакой переменной count в стеке больше нет.

    // Вместо нее создается объект класса DisplayClass1 и

    // используется его поле count.

    // В результате переменная count значимого типа располагается

    // не в стеке, а в управляемой куче и не будет собрана сборщиком

    // мусора до тех пор, пока будут оставаться ссылки на

    // делегат, созданный этим методом

    DisplayClass1 c1 = new DisplayClass1();

    c1.message = "Inside anonymous method";

    c1.count = 0;

    Action action = new Action(c1.ActionMethod);

    return action;

}

// Этот класс будет сгенерирован компилятором.

// Настоящее его имя будет иметь вид типа "<>c\_\_DisplayClass1".

// Таким образом компилятор обеспечивает невозможность коллизий

// имен, поскольку пользователь самостоятельно не сможет создать

// класс имя которого будет содержать символы "<>".

private sealed class DisplayClass1 : System.Object

{

    public DisplayClass1()

    {}

    // Имя этого метода тоже будет не столь благозвучным.

    // По тем же причинам (ради исключение

    // коллизий), имя это метода будет примерно

    // таким: "<CreateAction>b\_\_0".

    public void ActionMethod()

    {

        count++;

        Console.WriteLine("{0}. Count = {1}", message, count);

    }

    public int count;

}

static void Main(string[] args)

{

    var action = CreateAction();

    action();

    action();

}

**ПРИМЕЧАНИЕ**  
Приведенные “детали реализации” предназначены лишь для понимания поведения замыканий и могут изменяться от версии компилятора к версии. Все, на что может рассчитывать разработчик описано в спецификации языка C#.

Захваченные внешние переменные значимого типа (как локальные переменные, так и параметры, передаваемые по значению) больше не располагаются в стеке, вместо этого используется поле экземпляра сгенерированного класса.

**Поведение “захваченных” переменных**

Еще одной важной особенностью реализации замыканий в языке C# является то, что при замыкании осуществляется *захват переменной*, а не *захват значения переменной*, которое было перед определением делегата. И хотя к этому выводу вполне можно было прийти самостоятельно анализируя внутреннюю реализацию замыканий, я решил остановиться на этом более подробно. Давайте рассмотрим пример:

string str = "Initial value"; //1

Action action = ()=> //2

{

    Console.WriteLine(str); //3

    str = "Modified by closure";

};

str = "After delegate creation"; //4

action(); //5

Console.WriteLine(str); //6

Результат выполнения:  
After delegate creation  
Modified by closure

В этом фрагменте, вместо локальной переменной *str* используется поле анонимного типа, которое инициализируется значением “Initial value”. Далее, при создании анонимного метода, значение этой переменной никак не изменяется, а изменяется только в строке (4). Поскольку в этом фрагменте кода используется общий экземпляр класса string, то во время вызова делегата, значение строки уже будет не “Initial value”, а “After delegate creation”, а после завершения вызова анонимного метода, в строке (6), переменная *str* будет содержать значение “Modified by closure”.

**Захват переменных цикла**

Вооружившись знаниями о внутреннем устройстве замыканий и не забывая о том, что при замыкании осуществляется захват переменной, а не ее значения, нам осталось разобраться с последним примером: захватом переменных цикла.

var funcs = new List<Func<int>>();

for (int i = 0; i < 3; i++)

{

    funcs.Add(() => i);

}

foreach (var f in funcs)

    Console.WriteLine(f());

Запустив на выполнение этот фрагмент кода, вы получите следующий результат:  
3  
3  
3

Замыкания на переменные цикла зачастую приводят к очень неприятным последствиям, поскольку в этом случае поведение кода начинает отличаться от интуитивно понятного. Данный результат обусловлен двумя причинами: (1) при замыканиях осуществляется захват переменных, а не значений переменных и (2) в приведенном фрагменте кода, существует один экземпляр переменной *i*, который изменяется на каждой итерации цикла, а не создается новый экземпляр на каждой итерации. Сложив эти два пункта вместе мы получим, что будет создан только один объект анонимного типа (поскольку создается столько объектов анонимного типа, сколько *экземпляров* переменных захватывает анонимный метод) и за пределами цикла, мы будем использовать единственный экземпляр переменной *i*, который к тому времени будет равен трем.

Исправить ситуацию весьма просто:

var funcs = new List<Func<int>>();

for (int i = 0; i < 3; ++i)

{

**int tmp = i;**

    funcs.Add(() => **tmp**);

}

foreach (var f in funcs)

    Console.WriteLine(f());

В этом случае, на каждой итерации цикла будет осуществляться захват нового экземпляра переменной *tmp*, в результате чего, каждый экземпляр делегата будет ссылаться на свой собственный экземпляр анонимного класса, каждый из которых будет содержать собственную копию переменной *tmp*, значение которой будет соответствовать текущему значению переменной цикла. В этом случае вывод будет ожидаемым:

0  
1  
2

**ПРИМЕЧАНИЕ**  
Если вас интересует, почему осуществляется захватывают именно переменных, а не их значений или почему разработчики языка не добавили специальный синтаксис для возможности выбора поведения, я рекомендую почитать сообщения Эрика Липперта (Eric Lippert) [“О вреде замыканий на переменных цикла”](http://blogs.msdn.com/ruericlippert/archive/2009/11/12/9983705.aspx) и [“Замыкания на переменных цикла. Часть 2”](http://blogs.msdn.com/ruericlippert/archive/2009/11/16/9984832.aspx).

**Заключение**

Может показаться, что все эти знания о внутреннем устройстве и поведении  замыканий совершенно ни к чему простому программисту, который работает на императивных языках программирования, и который не стремится ко всей этой популярной ныне функциональщине. Но хотите вы этого или нет, замыкания являются частью языка программирования C# и даже если вы лично не будете использовать их в своей повседневной практике, то вы все равно обязательно столкнетесь с ними читая или изменяя код других разработчиков. А раз встречи с ними не избежать, то лучше подготовиться заранее и четко понимать, какие существуют преимущества, недостатки и особенности использования замыканий в вашем любимом языке программирования.

[Отправить по электронной почте](https://www.blogger.com/share-post.g?blogID=8596733192274108952&postID=7109279252756523623&target=email)[Написать об этом в блоге](https://www.blogger.com/share-post.g?blogID=8596733192274108952&postID=7109279252756523623&target=blog)[Опубликовать в Twitter](https://www.blogger.com/share-post.g?blogID=8596733192274108952&postID=7109279252756523623&target=twitter)[Опубликовать в Facebook](https://www.blogger.com/share-post.g?blogID=8596733192274108952&postID=7109279252756523623&target=facebook)[Поделиться в Pinterest](https://www.blogger.com/share-post.g?blogID=8596733192274108952&postID=7109279252756523623&target=pinterest)