**Ключевое слов lock**

Ключевое слово lock используется при создании многопоточных программ. Подробнее

оно рассматривается в главе 23, где речь пойдет о многопоточном программировании.

Но ради полноты изложения ниже приведено краткое описание этого ключевого

слова.

Программа на C# может состоять из нескольких *потоков исполнения.* В этом случае

программа считается *многопоточной,* и отдельные ее части выполняются параллельно,

т.е. одновременно и независимо друг от друга. В связи с такой организацией программы

возникает особого рода затруднение, когда два потока пытаются воспользоваться

ресурсом, которым можно пользоваться только по очереди. Для разрешения этого затруднения

можно создать *критический раздел кода,* который будет одновременно выполняться

одним и только одним потоком. И это делается с помощью ключевого слова

lock. Ниже приведена общая форма этого ключевого слова:

lock(obj) {

// критический раздел кода

}

где obj обозначает объект, для которого согласуется блокировка кода. Если один поток

уже вошел в критический раздел кода, то второму потоку придется ждать до тех пор,

пока первый поток не выйдет из данного критического раздела кода. Когда же первый

поток покидает критический раздел кода, блокировка снимается и предоставляется

второму потоку. С этого момента второй поток может выполнять критический раздел

кода.

**Ключевое слово readonly**

Отдельное поле можно сделать доступным в классе только для чтения, объявив его

как readonly. Значение такого поля можно установить только с помощью инициализатора,

когда оно объявляется или же когда ему присваивается значение в конструкторе.

После того как значение доступного только для чтения поля будет установлено,

оно не подлежит изменению за пределами конструктора. Следовательно, поле типа

readonly удобно для установки фиксированного значения с помощью конструктора.

Такое поле можно, например, использовать для обозначения размера массива, который

часто используется в программе. Допускаются как статические, так и нестатические

поля типа readonly.

(***glava20\_16***)

class MyClass

{

public static readonly int SIZE = 10;

}

class DemoReadOnly

{

static void Main()

{

int[] source = new int[MyClass.SIZE];

int[] target = new int[MyClass.SIZE];

for (int i = 0; i < MyClass.SIZE; i++)

source[i] = i;

Console.WriteLine();

for (int i = MyClass.SIZE - 1, j = 0; i > 0; i--, j++)

target[j] = source[i];

foreach (int i in target)

Console.Write(i + " ");

Console.WriteLine();

//MyClass.SIZE = 100;//Error!!

}

}

В данном примере поле MyClass.SIZE инициализируется значением 10. После

этого его можно использовать, но не изменять. Для того чтобы убедиться в этом, удалите

символы комментария в начале последней строки приведенного выше кода и попробуйте

скомпилировать его. В итоге вы получите сообщение об ошибке.

**Ключевые слова const и volatile**

Ключевое слово, или модификатор, const служит для объявления полей и локальных

переменных, которые нельзя изменять. Исходные значения таких полей и переменных

должны устанавливаться при их объявлении. Следовательно, переменная с

модификатором const, по существу, является константой. Например, в следующей

строке кода:

const int i = 10;

создается переменная i типа const и устанавливается ее значение 10. Поле типа const

очень похоже на поле типа readonly, но все же между ними есть отличие. Если поле

типа readonly можно устанавливать в конструкторе, то поле типа const — нельзя.

Ключевое слово, или модификатор, volatile уведомляет компилятор о том, что

значение поля может быть изменено двумя или более параллельно выполняющимися

потоками. В этой ситуации одному потоку может быть неизвестно, когда поле было

изменено другим потоком. И это очень важно, поскольку компилятор C# будет автоматически

выполнять определенную оптимизацию, которая будет иметь результат лишь

в том случае, если поле доступно только одному потоку. Для того чтобы подобной

оптимизации не подвергалось общедоступное поле, оно объявляется как volatile.

Этим компилятор уведомляется о том, что значение поля типа volatile следует получать

всякий раз, когда к нему осуществляется доступ.

**Оператор using**

Помимо рассматривавшейся ранее *директивы* using, имеется вторая форма ключевого

слова using в виде *оператора.* Ниже приведены две общие формы этого оператора:

using (obj) {

// использовать объект obj

}

using (тип obj = инициализатор) {

// использовать объект obj

}

где *obj* является выражением, в результате вычисления которого должен быть получен

объект, реализующий интерфейс System.IDisposable. Этот объект определяет

переменную, которая будет использоваться в блоке оператора using. В первой форме

объект объявляется вне оператора using, а во второй форме — в этом операторе. По

завершении блока оператора using для объекта *obj* вызывается метод Dispose(),

определенный в интерфейсе System.IDisposable. Таким образом, оператор using

предоставляет средства, необходимые для автоматической утилизации объектов, когда

они больше не нужны. Не следует, однако, забывать, что оператор using применяется

только к объектам, реализующим интерфейс System.IDisposable.

В приведенном ниже примере демонстрируются обе формы оператора using.

(Пример какое – то говно и шутка, он полностью списан из книги)

(***glava20\_17***)

class UsingDemo

{

static void Main()

{

try

{

StreamReader sr = new StreamReader("test.txt");

using (sr) { }

}

catch (IOException exc)

{ }

try

{

//create object streamreader in using

using (StreamReader sr2 = new StreamReader("test.txt"))

{

}

}

catch (IOException exc) { }

}

}

В данном примере интерфейс IDisposable реализуется в классе StreamReader

(посредством его базового класса TextReader). Поэтому он может использоваться в

операторе using. По завершении этого оператора автоматически вызывается метод

Dispose() для переменной потока, закрывая тем самым поток.

Как следует из приведенного выше примера, оператор using особенно полезен для

работы с файлами, поскольку файл автоматически закрывается по завершении блока

этого оператора, даже если он и завершается исключением. Таким образом, закрытие

файла с помощью оператора using зачастую упрощает код обработки файлов. Разумеется,

применение оператора using не ограничивается только работой с файлами.

В среде .NET Framework имеется немало других ресурсов, реализующих интерфейс

IDisposable. И всеми этими ресурсами можно управлять с помощью оператора

using.

**Ключевое слово extern**

Ключевое слово extern находит два основных применения. Каждое из них рассматривается

далее по порядку.

**Объявление внешних методов**

Оно обозначает, что метод предоставляется в неуправляемом коде, который

не является составной частью программы. Иными словами, метод предоставляется

внешним кодом.

Для того чтобы объявить метод как внешний, достаточно указать в самом начале его

объявления модификатор extern. Таким образом, общая форма объявления внешнего

метода выглядит следующим образом.

extern возвращаемый\_тип имя\_метода(список\_аргументов);

Обратите внимание на отсутствие фигурных скобок.

В данном варианте ключевое слово extern нередко применяется вместе с атрибутом

DllImport, обозначающим библиотеку DLL, в которой содержится внешний

метод. Атрибут DllImport принадлежит пространству имен System.Runtime.

InteropServices. Он допускает несколько вариантов, но, как правило, достаточно

указать лишь имя библиотеки DLL, в которой содержится внешний метод. Вообще говоря,

внешние методы следует программировать на С. (Если же это делается на С++, то

имя внешнего метода может быть изменено в библиотеке DLL путем дополнительного

оформления типов.)

Для того чтобы стало понятнее, как пользоваться внешними методами, обратимся

к примеру конкретной программы, состоящей из двух файлов. Ниже приведен исходный

код С из первого файла ExtMeth.с, где определяется метод AbsMax().

#include <stdlib.h>

int \_\_declspec(dllexport) AbsMax(int a, int b)

{

return abs(a) < abs(b) ? abs(b) : abs(a);

}

В методе AbsMax() сравниваются абсолютные значения двух его параметров

и возвращается самое большое из них. Обратите внимание на обозначение \_

declspec(dllexport). Это специальное расширение языка С для программных средств корпорации Microsoft. Оно уведомляет компилятор о необходимости экспортировать метод AbsMax() из библиотеки DLL, в которой он содержится.

(далее идет какой то очередной пример с библиотекой, сейчас нет времени чтобы создавать проект и разбираться с этим)

**Объявление псевдонима внешней сборки**

Во втором применении ключевое слово extern предоставляет псевдоним для

внешней сборки, что полезно в тех случаях, когда в состав программы включаются две

отдельные сборки с одним и тем же именем элемента. Так, если в сборке test1 содержится

класс MyClass, а в сборке test2 класс с таким же именем, то при обращении к

классу по этому имени в одной и той же программе может возникнуть конфликт.

Для разрешения подобного конфликта необходимо создать псевдоним каждой

сборки. Это делается в два этапа. На первом этапе нужно указать псевдонимы, используя

параметр компилятора /r, как в приведенном ниже примере. (я хз что означает следующий код, автор книги курнул, и как все авторы книг считает, что все кто читает книги – слишком умные чтобы разжевывать какое то говно, вроде длл и тд, подробно)

/r:Asm1=test1

/r:Asm2=test2

А на втором этапе необходимо ввести операторы с ключевым словом extern, в которых

делается ссылка на указанные выше псевдонимы. Ниже приведена форма такого

оператора для создания псевдонима сборки.

extern alias имя\_сборки;

Если продолжить приведенный выше пример, то в программе должны появиться

следующие строки кода.

extern alias Asm1;

extern alias Asm2;

Теперь оба варианта класса MyClass будут доступны в программе по соответствующему

псевдониму.