Управляющие операторы и методы   
В языке программирования C# существуют специальные операторы, которые в зависимости от вычисляемых значений выражений позволяют управлять ходом выполнения программы, эта лекция рассказывает именно о них http://www.intuit.ru/img/empty.gifhttp://www.intuit.ru/img/empty.gifhttp://www.intuit.ru/img/empty.gif

Основным средством реализации функциональности класса являются методы. Методы строятся из предложений-операторов, в ходе выполнения которых производятся вычисления составляющих операторы выражений. В языке программирования C# существуют специальные операторы, которые в зависимости от вычисляемых значений выражений позволяют управлять ходом выполнения программы.

В этой главе обсуждаются управляющие операторы и проблемы, связанные с определением и вызовом методов.

### Управляющие операторы

Управляющие операторы применяются в рамках методов. Они определяют последовательность выполнения операторов в программе и являются основным средством реализации алгоритмов.

Различаются следующие категории управляющих операторов:

* Операторы выбора. Вводятся ключевыми словами if, if ... else ..., switch.
* Итеративные операторы. Вводятся ключевыми словами while, do ... while, for, foreach.
* Операторы перехода (в рамках методов). Вводятся ключевыми словами goto, break, continue.

#### if, if ... else ...

После ключевого слова if располагается взятое в круглые скобки условное выражение (булево выражение), следом за которым располагается оператор (блок операторов) произвольной сложности.

Далее в операторе if ... else ... после ключевого слова else размещается еще один оператор.

В силу того, что в C# отсутствуют предопределенные алгоритмы преобразования значений к булевскому типу, условное выражение должно быть выражением типа bool – переменной, константой или выражением на основе операций сравнения и логических операций.

В соответствии с синтаксисом условного оператора в части else (если таковая имеется) располагается блок операторов. Частный случай оператора – оператор if.

Какой бы сложности ни были составляющие части if ... else ... – это всего лишь ДВЕ равноправных части одного оператора. "Каскад" в if ... else ... – это всего лишь оператор (блок), содержащий вхождение if или if ... else ... операторов.

if (...)

{

}

else

{

if (...)

{

}

else

{

}

}

переписывается

if (...)

{

}

else

if (...)

{

}

else

{

}

**Вложенный (Nesting)** if. Оператор if часто сам в свою очередь является условным оператором произвольной сложности.

И в этом случае оператор if ... else ... включает ДВЕ части.

if (...)

{

if (...)

{

}

else

{

}

}

else

{

}

переписывается как

if (...) if (...)

{

}

else

{

}

else

{

}

Главное – не перепутать соответствующие части оператора if ... else ... .

Еще одно важное замечание связано с использованием "простых" (не блоков) операторов. Невозможно построить оператор if ... else ... на основе одиночного оператора объявления:

if (true) int XXX = 125;

if (true) int XXX = 125; else int ZZZ = 10;

Такие конструкции воспринимаются как ошибочные. Одиночный оператор в C# — это не блок, и ставить в зависимость от условия (пусть даже всегда истинного) такое ответственное дело, как создание объекта, здесь не принято.

Совсем другое дело – при работе с блоками операторов!

if (true) {int XXX = 125;}

if (true) {int XXX = 125;} else {int ZZZ = 10;}

Даже в таких примитивных блоках определена своя область видимости, и создаваемые в них объекты, никому не мешая, существуют по своим собственным правилам.

#### switch

При описании синтаксиса оператора switch использована нотация Бэкуса-Наура (БНФ):

ОператорSWITCH ::= switch (switchВыражение){switchБлок}

switchВыражение ::= Выражение

switchВыражение может быть либо целочисленным, либо символьным. switchБлок не может быть пустым.

switchБлок ::= СписокCaseЭлементов

СписокCaseЭлементов ::= [СписокCaseЭлементов] CaseЭлемент

CaseЭлемент ::=

СписокРазделенныхМеток [СписокОператоров] ОператорТерминатор

СписокРазделенныхМеток ::= [СписокРазделенныхМеток] Метка РазделительМетки

РазделительМетки ::= :

Метка ::= case КонстантноеВыражение | default

ОператорТерминатор ::= breakОператор | gotoОператор | return [Выражение];

breakОператор ::= break;

gotoОператор ::= goto Метка;

Таким образом, минимальный switch -оператор имеет следующий вид:

int val;

::::::::::

switch (val)

{

default: break; // Список операторов пуст.

}

Более сложные образования:

int val;

::::::::::

switch (val)

{

case 0: break;

}

::::::::::

switch (val)

{

case 0: break;

default: break; // Список операторов пуст.

}

::::::::::

switch (val)

{

default: ... break; // default – равноправный элемент switch-оператора.

// может располагаться в любом месте switch-блока, поскольку

// НЕ БЛОКИРУЕТ (!!!) выполнение операторов всех нижележащих CaseЭлементов.

case 100: ... break;

}

::::::::::

switch (val)

{

default: // default НЕ БЛОКИРУЕТ выполнение операторов

// всех нижележащих CaseЭлементов.

// Операторы, входящие в CaseЭлемент под меткой 100, выполняются

// ПРИ ЛЮБЫХ значениях переменной val, отличных от 1 и 10.

case 100: ... break;

case 1: ... break;

case 10: ... break;

}

Подтверждение равноправия CaseЭлемента с меткой default. При выполнении менять значение switchВыражения. Осознать происходящее:

using System;

namespace OperatorsTest

{

class Program

{

static void Main(string[] args)

{

int i = 1; // = 500; = 125; = 100; = 250;

switch (i)

{

case 15:

default:

case 100:

Console.WriteLine("default: Ha-Ha-Ha");

break;

case 1:

case 2:

Console.WriteLine("1: 2: Ha-Ha-Ha");

break;

case 125:

Console.WriteLine("125: Ha-Ha-Ha! Ha-Ha-Ha");

break;

case 250:

Console.WriteLine("250: Ha-Ha-Ha! Ha-Ha-Ha");

break;

}

}

}

}

Поскольку при выполнении модуля выбор списков операторов для выполнения в рамках CaseЭлемента определяется значением константных выражений в case -метках, константные выражения ВСЕХ меток данного switchБлока должны различаться ПО СВОИМ ЗНАЧЕНИЯМ.

Следующий пример некорректен. Константные выражения в списках разделенных меток CaseЭлемента

case 1+1: case 2: ... break;

различаются ПО ФОРМЕ (1+1 и 2), а НЕ ПО ЗНАЧЕНИЮ!

По тем же причинам в рамках switch -оператора может быть не более одного вхождения метки default.

Списки операторов в CaseЭлементах НЕ могут включать операторы объявления. switchОператор строится на основе ОДНОГО блока, разделяемого метками на фрагменты, выполнение которых производится в зависимости от значений константных выражений в case -метках. Разрешение объявления констант и переменных в CaseЭлементе означает риск обращения к ранее необъявленной переменной:

switch (val)

{

case 0:

int XXX = 100; // Нельзя!

break;

case 1:

XXX += 125;

break;

}

Каждый CaseЭлемент в обязательном порядке ЗАВЕРШАЕТСЯ оператором-терминатором, который является ОБЩИМ для ВСЕХ операторов данного CaseЭлемента:

int val, XXX;

:::::

switch (val)

{

case 0:

if (XXX == 25) {XXX \*= –1; break;}

else XXX = 17;

goto default; // Терминатор.

case 1:

XXX += 125;

break; // Терминатор.

default:

return XXX; // Терминатор.

}

#### while

ОператорWHILE ::= while (УсловиеПродолжения) Оператор

УсловиеПродолжения ::= БулевоВыражение

Здесь

Оператор ::= Оператор

::= БлокОператоров

Правило выполнения этого итеративного оператора состоит в следующем: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Невозможно построить операторWHILE на основе одиночного оператора объявления. Оператор

while (true) int XXX = 0;

с самого первого момента своего существования (еще до начала трансляции!) сопровождается предупреждением:

Embedded statement cannot be a declaration or labeled statement.

#### do ... while

ОператорDOWHILE ::= do Оператор while (УсловиеПродолжения)

УсловиеПродолжения ::= БулевоВыражение

Оператор ::= Оператор

::= БлокОператоров

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения оператора.

#### for

ОператорFOR ::=

for ([ВыраженияИнициализации];[УсловиеПродолжения];[ВыраженияШага]) Оператор

ВыраженияИнициализации ::= СписокВыражений

СписокВыражений ::= [СписокВыражений ,] Выражение

УсловиеПродолжения ::= БулевоВыражение

ВыраженияШага ::= [СписокВыражений ,] Выражение

Здесь

Оператор ::= Оператор

::= БлокОператоров

ВыраженияИнициализации, УсловиеПродолжения, ВыраженияШага в заголовке оператора цикла for могут быть пустыми. Однако наличие пары символов ' ; ' в заголовке цикла for обязательно.

Список выражений представляет собой разделенную запятыми последовательность выражений.

Следует иметь в виду, что оператор объявления также строится на основе списка выражений (выражений объявления), состоящих из спецификаторов типа, имен и, возможно, инициализаторов. Этот список завершается точкой с запятой, что позволяет рассматривать список выражений инициализации как самостоятельный оператор в составе оператора цикла for. При этом область видимости имен переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла. Это значит, что переменные, объявленные в операторе инициализации данного оператора цикла, НЕ МОГУТ быть использованы непосредственно после оператора до конца блока, содержащего этот оператор. А следующие друг за другом в рамках общего блока операторы МОГУТ содержать в заголовках одни и те же выражения инициализации.

операторFOR также невозможно построить на основе одиночного оператора объявления.

#### foreach

ОператорFOREACH ::=

foreach (ОбъявлениеИтератора in ВыражениеIN) Оператор

ОбъявлениеИтератора ::= ИмяТипа Идентификатор

ВыражениеIN ::= Выражение

Оператор ::= Оператор

::= БлокОператоров

ИмяТипа – обозначает тип итератора.

identifier – обозначает переменную, которая представляет элемент коллекции.

ВыражениеIN – объект, представляющий массив или коллекцию.

Этим оператором обеспечивается повторение множества операторов, составляющих тело цикла, для каждого элемента массива или коллекции. После перебора ВСЕХ элементов массива или коллекции и применения множества операторов для каждого элемента массива или коллекции, управление передается следующему за Оператором FOREACH оператору (разумеется, если таковые имеются).

Область видимости имен переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла:

int[] array = new int[10]; // Объявили и определили массив

foreach (int i in array) {/\*:::::\*/}; // Для каждого элемента массива надо сделать...

Специализированный оператор, приспособленный для работы с массивами и коллекциями. Обеспечивает повторение множества (единичного оператора или блока операторов) операторов для КАЖДОГО элемента массива или коллекции. Конструкция экзотическая и негибкая. Предполагает выполнение примитивных последовательностей действий над массивами и коллекциями (начальная инициализация или просмотр ФИКСИРОВАННОГО количества элементов). Действия, связанные с изменениями размеров и содержимого коллекций, в рамках этого оператора могут привести к непредсказуемым результатам.

#### goto, break, continue

goto в операторе switch уже обсуждалось.

Второй вариант использования этого оператора — непосредственно тело метода.

Объявляется метка (правильнее "оператор с меткой"). Оператор может быть пустым. Метка-идентификатор отделяется от оператора двоеточием. В качестве дополнительного разделителя могут быть использованы пробелы, символы табуляции и перехода на новую строку. Метка, как и любое другое имя, подчиняется правилам областей видимости. Она видна в теле метода только в одном направлении: из внутренних (вложенных) блоков. Поэтому оператор перехода

goto ИмяПомеченногоОператора;

позволяет ВЫХОДИТЬ из блоков, но не входить в них.

Обычно об этом операторе говорится много нехороших слов как о главном разрушителе структурированного программного кода, и его описание сопровождается рекомендациями к его НЕИСПОЛЬЗОВАНИЮ.

Операторы

break;

и

continue;

применяются как вспомогательные средства управления в операторах цикла.

### Методы

В C# методы определяются в рамках объявления класса. Методы (функции) являются членами класса и определяют функциональность объектов — членов класса (нестатические методы – методы объектов) и непосредственно функциональность самого класса (статические методы – методы класса).

Метод может быть объявлен, и метод может быть вызван. Поэтому различают объявление метода (метод объявляется в классе) и вызов метода (выражение вызова метода располагается в теле метода).

Различают статические (со спецификатором static ) и нестатические методы (объявляются без спецификатора).

#### Синтаксис объявления метода

ОбъявлениеМетода ::= ЗаголовокМетода ТелоМетода

ЗаголовокМетода ::= [СпецификаторМетода]

ТипВозвращаемогоЗначения

Имя

([СписокПараметров])

СпецификаторМетода ::= СпецификаторДоступности

::= new

::= static

::= virtual

::= sealed

::= override

::= abstract

::= extern

СпецификаторДоступности ::= public

::= private

::= protected

::= internal

::= protected internal

ТипВозвращаемогоЗначения ::= void | ИмяТипа

ТелоМетода ::= БлокОператоров

::= ;

Имя ::= Идентификатор

СписокПараметров ::= [СписокПараметров ,] ОбъявлениеПараметра

ОбъявлениеПараметра ::= [СпецификаторПередачи] ИмяТипа ИмяПараметра

::= [СпецификаторСписка] ИмяТипаСписка ИмяПараметра

СпецификаторПараметра ::= СпецификаторПередачи | СпецификаторСписка

СпецификаторПередачи ::= ref | out

СпецификаторСписка ::= params

ИмяТипаСписка ::= ИмяТипа[]

Листинг 3.1.

Тело метода может быть пустым! В этом случае за заголовком метода располагается точка с запятой.

Класс, объявление которого содержит только объявления статических методов, называется статическим классом и может объявляться со спецификатором static:

public static class XXX

{

static int f1(int x)

{

return 1;

}

static int f2(int x)

{

return 2;

}

}

#### Вызов метода

Выражение вызова метода — это всего лишь выражение, составная часть оператора (предложения C#). Оно располагается в блоке операторов — в теле метода. Выражению вызова метода может предшествовать выражение, определяющее принадлежность метода классу или объекту.

Статические методы определяют функциональность класса. Для статических методов, принадлежащих другому классу, этим выражением может быть имя класса, содержащего объявление данного метода. Таким образом, статические методы вызываются от имени класса, в котором они были объявлены.

Нестатические методы определяют поведение конкретных объектов-представителей класса и потому вызываются "от имени" объекта-представителя класса, содержащего объявление вызываемого метода. Указание на конкретный объект — это всего лишь выражение, обеспечивающее ссылку на данный объект. Таковым действительно может быть имя объекта, выражение this (которое вообще может быть опущено), выражение индексации и другие выражения ссылочного типа.

Выражение вызова метода можно считать точкой вызова метода. В точке вызова управление передается вызываемому методу. После выполнения последнего оператора в теле вызываемого метода управление возвращается к точке вызова. Если метод возвращает значения, выражение вызова принимает соответствующее значение. В момент получения возвращаемого значения точка вызова становится точкой возврата.

### Обработка исключений

Пусть в классе объявляются методы A и B.

При этом из метода A вызывается метод B, который выполняет свою работу, возможно, возвращает результаты. В теле метода A есть точка вызова метода B и точка возврата, в которой оказывается управление после успешного возвращения из метода B.

Если все хорошо, метод A, возможно, анализирует полученные результаты и продолжает свою работу непосредственно из точки возврата.

Если при выполнении метода B возникла исключительная ситуация (например, целочисленное деление на 0), возможно, что метод A узнает об этом, анализируя возвращаемое из B значение. Таким может быть один из сценариев "обратной связи", при котором вызывающий метод узнает о результатах деятельности вызываемого метода.

Недостатки этого сценария заключаются в том, что:

* метод B может в принципе не возвращать никаких значений;
* среди множества возвращаемых методом B значений невозможно выделить подмножество значений, которые можно было бы воспринимать как уведомление об ошибке;
* работа по подготовке уведомления об ошибке требует неоправданно больших усилий.

Решение проблемы состоит в том, что в среде выполнения поддерживается модель обработки исключений, основанная на понятиях объектов исключения и защищенных блоков кода. Следует отметить, что схема обработки исключений не нова и успешно реализована во многих языках и системах программирования.

Некорректная ситуация в ходе выполнения программы (деление на нуль, выход за пределы массива) рассматривается как исключительная ситуация, и метод, в котором она произошла, реагирует на нее ГЕНЕРАЦИЕЙ ИСКЛЮЧЕНИЯ, а не обычным возвращением значения, пусть даже изначально ассоциированного с ошибкой.

Среда выполнения создает объект для представления исключения при его возникновении. Одновременно с этим прерывается обычный ход выполнения программы. Происходит так называемое разматывание стека, при котором управление НЕ оказывается в точке возврата, и если ни в одном из методов, предшествующих вызову, не было предпринято предварительных усилий по ПЕРЕХВАТУ ИСКЛЮЧЕНИЯ, приложение аварийно завершается.

Можно написать код, обеспечивающий корректный перехват исключений, можно создать собственные классы исключений, получив производные классы из соответствующего базового исключения.

Все языки программирования, использующие среду выполнения, обрабатывают исключения одинаково. В каждом языке используется форма try/catch/finally для структурированной обработки исключений.

Следующий пример демонстрирует основные принципы организации генераторов и перехватчиков исключений.

using System;

// Объявление собственного исключения.

// Наследуется базовый класс Exception.

public class xException:Exception

{

// Собственное исключение имеет специальную строчку,

// и в этом ее отличие.

public string xMessage;

// Кроме того, в поле ее базового элемента

// также фиксируется особое сообщение.

public xException(string str):base("xException is here...")

{

xMessage = str;

}

}

// Объявление собственного исключения.

// Наследуется базовый класс Exception.

public class MyException:Exception

{

// Собственное исключение имеет дополнительную

// специальную строчку для кодирования информации

// об исключении.

public string MyMessage;

// Кроме того, в поле ее базового элемента

// также фиксируется особое сообщение.

public MyException(string str):base("MyException is here...")

{

MyMessage = str;

}

}

public class StupidCalcule

{

public int Div(int x1, int x2)

{

// Вот здесь метод проверяет корректность операндов и с помощью оператора

// throw возбуждает исключение.

if (x1 != 0 && x2 == 0)

throw new Exception("message from Exception: Incorrect x2!");

else if (x1 == 0 && x2 == 0)

throw new MyException("message from MyException: Incorrect x1

&& x2!");

else if (x1 == –1 && x2 == –1)

throw new xException("message from xException: @#$%\*&^???");

// Если же ВСЕ ХОРОШО, счастливый заказчик получит ожидаемый результат.

return (int)(x1/x2);

}

}

public class XXX

{

public static void Main()

{

int ret;

StupidCalcule sc = new StupidCalcule();

// Наше приложение специально подготовлено к обработке исключений!

// Потенциально опасное место (КРИТИЧЕСКАЯ СЕКЦИЯ) ограждено

// (заключено в блок try)

try

{

// Вызов...

ret = sc.Div(–1,–1);

// Если ВСЕ ХОРОШО – будет выполнен оператор Console.WriteLine.

// Потом операторы блока finally, затем операторы за

// пределами блоков try, catch, finally.

Console.WriteLine("OK, ret == {0}.", ret.ToString());

}

catch (MyException e)

{

// Здесь перехватывается MyException.

Console.WriteLine((Exception)e);

Console.WriteLine(e.MyMessage);

}

// Если этот блок будет закомментирован –

// возможные исключения типа Exception и xException

// окажутся неперехваченными. И после блока

// finally приложение аварийно завершится.

catch (Exception e)

{

// А перехватчика xException у нас нет!

// Поэтому в этом блоке будут перехватываться

// все ранее неперехваченные потомки исключения Exception.

// Это последний рубеж.

// Еще один вариант построения последнего рубежа

// выглядит так:

// catch

//{

// Операторы обработки – сюда. Здесь в любом случае код,

// реализующий алгоритм последнего перехвата.

Console.WriteLine(e);

}

finally

{

// Операторы в блоке finally выполняются ВСЕГДА, тогда как

// операторы, расположенные за пределами блоков try, catch, finally,

// могут и не выполниться вовсе.

Console.WriteLine("finally block is here!");

}

// Вот если блоки перехвата исключения окажутся не

// соответствующими возникшему исключению – нормальное

// выполнение приложения будет прервано – и мы никогда не увидим

// этой надписи.

Console.WriteLine("Good Bye!");

}

}

Листинг 3.2.

### Работа с входным потоком. Предварительная информация

Информация, поступающая из входного потока, представляет собой символьные последовательности. Естественно, что область применения вводимой информации существенно ограничена. Арифметические вычисления требуют значений соответствующего (арифметического) типа.

Следующий пример демонстрирует способы преобразования поступающей через входной поток (ввод с клавиатуры) информации с целью получения значений арифметического типа:

using System;

namespace InOutTest00

{

public class InputTester

{

// Ввод арифметических значений типов int и float.

// Осуществляется в рамках бесконечного цикла, поскольку

// предполагает возможные ошибки при вводе.

// Выход из бесконечного цикла

// происходит только после прочтения из входного потока

// последовательности символов, которая может быть

// преобразована в значения типа int или float.

public int intIput()

{

string inputStr;

int val = 0;

for ( ; ; )

{

Console.Write("integer value, please >>> ");

// Строка символов читается с клавиатуры.

inputStr = Console.ReadLine();

try

{

// Попытка преобразования строки символов к типу int.

// В случае неудачи возникает исключение.

val = int.Parse(inputStr);

}

catch (Exception e)

{

// Перехват исключения.

// Сообщили об ошибке ввода.

Console.WriteLine("Input Error. {0} isn't integer", inputStr);

// Собственно информация об исключении.

Console.WriteLine("{0}", e.ToString());

// Повторяем попытку ввода символьной строки.

continue;

}

// Успешная попытка преобразования строки символов к типу int.

// Выход из цикла.

return val;

}

}

// Ввод значений типа float.

// Выход из бесконечного цикла

// происходит только после прочтения из входного потока

// последовательности символов, которая может быть

// преобразована в значение типа float.

// На правильность ввода влияют настройки языковых параметров,

// представления чисел, денежных единиц, времени и дат.

public float floatIput()

{

string inputStr;

float val;

for ( ; ; )

{

Console.Write("float value, please >>> ");

inputStr = Console.ReadLine();

try

{

// Приведение строки символов к значению типа float

// с использованием статических методов класса Convert.

val = System.Convert.ToSingle(inputStr);

}

catch (Exception e)

{

Console.WriteLine("Input Error!");

Console.WriteLine("{0}", e.ToString());

continue;

}

return val;

}

}

}

class Program

{

// Точка входа.

static void Main(string[] args)

{

// Создается объект – представитель класса InputTester.

InputTester it = new InputTester();

// Этот объект используется для корректного ВВОДА с клавиатуры:

// значений типа int...

int iVal = it.intIput();

// значений типа float...

float fVal = it.floatIput();

}

}

}

Листинг 3.3.

### Перегрузка методов

Имя метода и список типов параметров метода являются его важной характеристикой и называются СИГНАТУРОЙ метода. В C# методы, объявляемые в классе, идентифицируются по сигнатуре. Эта особенность языка позволяет объявлять в классе множество одноименных методов. Такие методы называются перегруженными, а деятельность по их объявлению – перегрузкой.

При написании программного кода, содержащего ВЫРАЖЕНИЯ ВЫЗОВА переопределенных методов, корректное соотнесение выражения вызова метода определяет метод, которому будет передано управление:

// Класс содержит объявление четырех одноименных методов

// с различными списками параметров.

class C1

{

void Fx(float key1)

{

return;

}

int Fx(int key1)

{

return key1;

}

int Fx(int key1, int key2)

{

return key1;

}

int Fx(byte key1, int key2)

{

return (int)key1;

}

static void Main(string[] args)

{

C1 c = new C1();

// Нестатические методы вызываются от имени объекта c.

// Передача управления соответствующему методу

// обеспечивается явными преобразованиями к типу.

c.Fx(Convert.ToSingle(1));

c.Fx(3.14F);

c.Fx(1);

c.Fx(1,2);

c.Fx((byte)10, 125);

}

}

Информация о типе возвращаемого значения при этом не учитывается, поскольку в выражении вызова возвращаемое значение метода может не использоваться вовсе.

### Способы передачи параметров при вызове метода

Известно два способа передачи параметров при вызове метода:

* по значению (в силу специфики механизма передачи параметров – только входные),
* по ссылке (входные и/или выходные).

Передача по значению – БЕЗ спецификаторов (для типов-значений этот способ предполагается по умолчанию). Параметр представляет собой локальную копию значения в методе. В теле метода это означенная переменная, которую можно использовать в методе наряду с переменными, непосредственно объявленными в этом методе. При этом изменение значения параметра НЕ влияет на значение параметра в выражении вызова.

Для организации передачи по ссылке параметра типа значения требуется явная спецификация ref. Для ссылочных типов передача параметра по ссылке предполагается по умолчанию. Спецификатор ref в этом случае не требуется, поскольку другого способа передачи параметра для ссылочных типов просто нет.

При передаче значения по ссылке также может использоваться спецификатор out. Этим спецификатором обозначают параметры, которым в методе присваиваются значения. Наличие в вызываемом методе выражения, обеспечивающего присвоение значения параметру со спецификатором out, обязательно.

Выражение вызова метода приобретает свое значение непосредственно по возвращении из вызываемого метода. Этим значением можно воспользоваться только в точке возврата.

Переданные методу "для означивания" параметры со спецификатором out сохраняют свои значения за точкой возврата.

Спецификация способа передачи параметра является основанием для перегрузки метода.

using System;

class XXX {

public int mmm;

}

class Class1

{//=============================================

static int i;

static void f1 (ref int x)

{

x = 125;

}

static void f1 (int x)

{

x = 0;

}

static void f1 (XXX par)

{

par.mmm = 125;

}

static void f1 (out XXX par)

{

par = new XXX(); // Ссылка на out XXX par ДОЛЖНА быть

// обязательно проинициализирована в теле

// метода НЕПОСРЕДСТВЕННО перед обращением к ней!

// Способ инициализации – любой! В том числе и созданием объекта!

// А можно и присвоением в качестве значения какой-либо другой ссылки.

par.mmm = 125;

}

// Для параметра типа значения спецификаторы out–ref

// неразличимы.

//static void f1 (out int x)

//{

// x = 125;

//}

static void Main(string[] args)

{//======================================

int a = 0;

f1(ref a);

//f1(out a);

f1(a);

XXX xxx = new XXX();

xxx.mmm = 0;

f1(xxx);

f1(ref xxx);

// По возвращении из метода это уже другая ссылка!

// Под именем xxx – другой объект.

}//======================================

}//=============================================

Листинг 3.4.

### Ссылка и ссылка на ссылку как параметры

На самом деле ВСЕ параметры ВСЕГДА передаются по значению. Это означает, что в области активации создается копия ЗНАЧЕНИЯ параметра. При этом важно, ЧТО копируется в эту область. Для ссылочных типов возможны два варианта:

* можно копировать адрес объекта (ссылка как параметр),
* можно копировать адрес переменной, которая указывает на объект (ссылка на ссылку как параметр).

В первом случае параметр обеспечивает изменение полей объекта. Непосредственное изменение значения этой ссылки (возможно, в результате создания новых объектов и присвоения значения новой ссылки параметру) означает лишь изменение значения параметра, который в данном случае играет роль обычной переменной, сохраняющей значение какого-то адреса (адреса ранее объявленного объекта).

Во втором случае параметр сохраняет адрес переменной, объявленной в теле вызывающего метода. При этом в вызываемом методе появляется возможность как изменения значений полей объекта, так и непосредственного изменения значения переменной.

Следующий программный код демонстрирует специфику передачи параметров:

using System;

namespace Ref\_RefRef

{

// Ссылка и ссылка на ссылку.

class WorkClass

{

public int x;

// Варианты конструкторов.

public WorkClass()

{

x = 0;

}

public WorkClass(int key)

{

x = key;

}

public WorkClass(WorkClass wKey)

{

x = wKey.x;

}

}

class ClassRef

{

static void Main(string[] args)

{

WorkClass w0 = new WorkClass();

Console.WriteLine("on start: {0}", w0.x); //\_: 0

f0(w0);

Console.WriteLine("after f0: {0}", w0.x); //0: 1

f1(w0);

Console.WriteLine("after f1: {0}", w0.x); //1: 1

f2(ref w0);

Console.WriteLine("after f2: {0}", w0.x); //2: 10

f3(ref w0);

Console.WriteLine("after f3: {0}", w0.x); //3: 3

// Еще один объект...

WorkClass w1 = new WorkClass(w0);

ff(w0, ref w1);

if (w0 == null) Console.WriteLine("w0 == null");

else Console.WriteLine("w0 != null"); // !!!

if (w1 == null) Console.WriteLine("w1 == null"); // !!!

else Console.WriteLine("w1 != null");

}

static void f0(WorkClass wKey)

{

wKey.x = 1;

Console.WriteLine(" in f0: {0}", wKey.x);

}

static void f1(WorkClass wKey)

{

wKey = new WorkClass(2);

Console.WriteLine(" in f1: {0}", wKey.x);

}

static void f2(ref WorkClass wKey)

{

wKey.x = 10;

Console.WriteLine(" in f2: {0}", wKey.x);

}

static void f3(ref WorkClass wKey)

{

wKey = new WorkClass(3);

Console.WriteLine(" in f3: {0}", wKey.x);

}

static void ff(WorkClass wKey, ref WorkClass refKey)

{

wKey = null;

refKey = null;

}

}

}

Листинг 3.5.

В результате выполнения программы в окошко консольного приложения выводится информация следующего содержания:

on start: 0

in f0: 1

after f0: 1

in f1: 2

after f1: 1

in f2: 10

after f2: 10

in f3: 3

after f3: 3

w0 != null

w1 == null

### Сравнение значений ссылок

Существуют следующие варианты:

* ссылка может быть пустой ( ref0 == null || ref1 != null );
* разные ссылки могут быть настроены на разные объекты ( ref0 != ref1 );
* разные ссылки могут быть настроены на один объект ( ref0 == ref1 );
* четвертого не дано. Отношение "больше-меньше" в условиях управляемой памяти не имеет никакого смысла.

### Свойства

Объявляемые в классе данные-члены обычно используются как переменные. Статические члены сохраняют значения, актуальные для всех объектов-представителей класса. Нестатические данные-члены сохраняют в переменных объекта информацию, актуальную для данного объекта.

Обращение к этим переменным производится с использованием точечной нотации, с явным указанием данного-члена, предназначенного для сохранения (или получения) значения.

В отличие от переменных, свойства не указывают на конкретные места хранения значений. Определение свойства в C# предполагает описание способов чтения и записи значений. Алгоритмы чтения и записи значений реализуются в виде блоков операторов, которые называются get accessor и set accessor.

Наличие accessor'ов определяет доступность свойства для чтения и записи. При обращении к значению свойства активизируется механизм чтения (управление передается в get accessor), при изменении значения активизируется механизм записи (управление передается в set accessor):

class TestClass

{

int xVal; // Переменная объекта.

// Свойство, обслуживающее переменную объекта.

// Предоставляет возможность чтения и записи значений

// поля xVal.

public int Xval

{

// Эти значения реализованы в виде блоков программного кода,

// обеспечивающих получение и изменение значения поля.

// get accessor.

get

{

// Здесь можно расположить любой код.

// Он будет выполняться после обращения к свойству для

// прочтения значения.

// Например, так. Здесь получается,

// что возвращаемое значение зависит от количества

// обращений к свойству.

xVal++;

return xVal;

}

set

{

// set accessor.

// Здесь можно расположить любой код.

// Он будет выполняться после обращения к свойству для

// записи значения.

xVal = value;

}

}

}

class Class1

{

static void Main(string[] args)

{

// Создали объект X.

TestClass X = new TestClass();

// Обратились к свойству для записи в поле Xval

// значения. Обращение к свойству располагается

// СЛЕВА от операции присвоения. В свойстве Xval

// будет активизирован блок кода set.

X.Xval = 100;

// Обратились к свойству для чтения из поля Xval

// значения. Обращение к свойству располагается

// СПРАВА от операции присвоения. В свойстве Xval

// будет активизирован блок кода get.

int q = X.Xval;

}

}

Листинг 3.6.

Поскольку объект для доступа к данным Get концептуально равнозначен чтению значения переменной, хорошим стилем программирования считается отсутствие заметных побочных эффектов при использовании объектов для доступа к данным Get.

Значение свойства не должно зависеть от каких-либо обстоятельств, в частности, от количества обращений к объекту. Если доступ к свойству порождает (как в нашем случае) заметный побочный эффект, свойство рекомендуется реализовывать как метод.

### Main в классе. Точка входа

Без статической функции (метода) Main невозможно построить выполняемую программу. Без явно обозначенной точки входа сборка не может выполняться.

В сборке можно помещать несколько классов. Каждый класс располагает собственным набором методов. В каждом классе могут находиться одноименные методы. В следующем примере объявляются три класса в одном пространстве имен. В каждом классе объявляется независимая точка входа и три (!) СТАТИЧЕСКИЕ функции Main (возможно и такое). Здесь главная проблема – при компиляции надо явным образом указать точку входа.

* Это можно сделать из командной строки при вызове компилятора. Например, так:

c:\ csc /main:Class1.Class3 Example1.cs

* Это можно сделать через диалог The Startup Object property среды разработки приложений, который обеспечивает спецификацию значений, явным образом НЕ ПРОПИСАННЫХ в проекте. Меню Проект – Свойства проекта, далее – General, Common Properties, <Projectname> Property Pages Dialog Box (Visual C#). В разделе Startup object надо раскрыть список классов и указать соответствующий класс.

Транслятор создаст сборку, в которой будет обеспечена передача управления соответствующей функции Main (одной из трех!):

using System;

namespace Example1

{

//===============================================

public class Class1

{

// Спецификатор public нужен здесь. Третий класс.

public class Class3

{

public static void Main()

{

string[] sss = new string[]{Class1.s.ToString(),"12345"};

Class1.Main(sss);

}

}

int d = 0;

public static int s;

static void Main(string[] args)

{

Class1 c1 = new Class1();

f1(c1);

c1.f2();

Class2 c2 = new Class2();

//c2.f2();

c2.f3();

string[] sss = new string[] {"qwerty", c1.ToString()};

Class2.Main(sss);

}

static void f1(Class1 x)

{

//x.s = 100;

s = 0;

Class1.s = 125;

x.d = 1;

//d = 100;

}

void f2()

{

s = 0;

Class1.s = 100;

//this.s = 5;

//Class1.d = 125;

this.d = 100;

d = 100;

}

}

//===============================================

class Class2

{

int d;

static int s;

public static void Main(string[] args)

{

Class1.Class3.Main();

Class2 c2 = new Class2();

f1(c2);

c2.f2();

//Class1.Main();

}

static void f1(Class2 x)

{

//x.s = 100;

s = 0;

Class2.s = 125;

x.d = 1;

//d = 100;

}

void f2()

{

s = 0;

Class1.s = 100;

//this.s = 5;

//Class1.d = 125;

this.d = 100;

d = 100;

}

public void f3()

{

s = 0;

Class1.s = 100;

//this.s = 5;

//Class1.d = 125;

this.d = 100;

d = 100;

}

}

//===============================================

}

Листинг 3.7.

Структура также может иметь свою точку входа!

using System;

namespace defConstructors

{

struct MyStruct

{

static void Main(string[] args)

{

Console.WriteLine("Ha-Ha-Ha");

}

}

}