*Исключительная ситуация,* или просто исключение, происходит во время выполнения. Используя подсистему обработки исключительных ситуаций в С#, можно обрабатывать структурированным и контролируемым образом ошибки, возникающие при выполнении программы. Главное преимущество обработки исключительных ситуаций заключается в том, что она позволяет автоматизировать получение большей части кода, который раньше приходилось вводить в любую крупную программу вручную для обработки ошибок. Так, если программа написана на языке программирования без обработки исключительных ситуаций, то при неудачном выполнении методов приходится возвращать коды ошибок, которые необходимо проверять вручную при каждом вызове метода. Это не только трудоемкий, но и чреватый ошибками процесс. Обработка исключительных ситуаций рационализирует весь процесс обработки ошибок, позволяя определить в программе блок кода, называемый *обработчиком исключений* и выполняющийся автоматически, когда возникает ошибка. Эго избавляет от необходимости проверять вручную, насколько удачно или неудачно завершилась конкретная операция либо вызов метода. Если возникнет ошибка, она будет обработана соответствующим образом обработчиком ошибок.

# Класс System.Exception

В C# исключения представлены в виде классов. Все классы исключений должны

быть производными от встроенного в C# класса Exception, являющегося частью пространства

имен System. Следовательно, все исключения являются подклассами класса

Exception.

К числу самых важных подклассов Exception относится класс SystemException.

Именно от этого класса являются производными все исключения, генерируемые исполняющей

системой C# (т.е. системой CLR). Класс SystemException ничего не добавляет

к классу Exception, а просто определяет вершину иерархии стандартных исключений.

В среде .NET Framework определено несколько встроенных исключений, являющихся

производными от класса SystemException. Например, при попытке выполнить

деление на нуль генерируется исключение DivideByZeroException. Как будет

показано далее в этой главе, в C# можно создавать собственные классы исключений,

производные от класса Exception.

# Основы обработки исключительных ситуаций

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых

слов: try, catch, throw и finally. Они образуют взаимосвязанную подсистему,

в которой применение одного из ключевых слов подразумевает применение другого.

На протяжении всей этой главы назначение и применение каждого из упомянутых

выше ключевых слов будет рассмотрено во всех подробностях. Но прежде необходимо

дать общее представление о роли каждого из них в обработке исключительных ситуаций.

Поэтому ниже кратко описан принцип их действия.

Операторы программы, которые требуется контролировать на появление исключений,

заключаются в блок try. Если внутри блока try возникает исключительная ситуация,

*генерируется* исключение. Это исключение может быть перехвачено и обработано

каким-нибудь рациональным способом в коде программы с помощью оператора, обозначаемого

ключевым словом catch. Исключения, возникающие на уровне системы,

генерируются исполняющей системой автоматически. А для генерирования исключений

вручную служит ключевое слово throw. Любой код, который должен быть непременно

выполнен после выхода из блока try, помещается в блок finally.

# Применение пары ключевых слов try и catch

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов

try и catch. Эти ключевые слова действуют совместно и не могут быть использованы

порознь. Ниже приведена общая форма определения блоков try/catch для обработки

исключительных ситуаций:

try {

// Блок кода, проверяемый на наличие ошибок.

}

catch (ExcepType1 exOb) {

// Обработчик исключения типа ExcepTypel.

}

catch (ExcepType2 exOb) {

// Обработчик исключения типа ExcepType2.

}

где *ЕхсерТуре —* это тип возникающей исключительной ситуации. Когда исключение

генерируется оператором try, оно перехватывается составляющим ему пару оператором

catch, который затем обрабатывает это исключение. В зависимости от типа

исключения выполняется и соответствующий оператор catch. Так, если типы генерируемого

исключения и того, что указывается в операторе catch, совпадают, то выполняется

именно этот оператор, а все остальные пропускаются. Когда исключение

перехватывается, переменная исключения *exOb* получает свое значение.

На самом деле указывать переменную *ехОb* необязательно. Так, ее необязательно

указывать, если обработчику исключений не требуется доступ к объекту исключения,

что бывает довольно часто. Для обработки исключения достаточно и его типа. Именно

поэтому во многих примерах программ, приведенных в этой главе, переменная *ехОb*

опускается.

# Простой пример обработки исключительной ситуации

Рассмотрим простой пример, демонстрирующий отслеживание и перехватывание

исключения. Как вам должно быть уже известно, попытка индексировать массив за его

границами приводит к ошибке. Когда возникает подобная ошибка, система CLR генерирует

исключение IndexOutOfRangeException, которое определено как стандартное

для среды .NET Framework. В приведенной ниже программе такое исключение

генерируется намеренно и затем перехватывается.

(***glava13\_1***)

class ExcDemo1

{

static void Main()

{

int[] nums = new int[4];

try

{

Console.WriteLine("Before exception block.");

//generate exception by index out of range

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

{

nums[i] = i;

Console.WriteLine("nums[{0}]: {1}", i, nums[i]);

}

Console.WriteLine("Error.");

}

catch(IndexOutOfRangeException)

{

//catch exception

Console.WriteLine("Index out of range of array!");

}

Console.WriteLine("After exception block");

}

}

В данном примере массив nums типа int состоит из четырех элементов. Но в цикле

for предпринимается попытка проиндексировать этот массив от 0 до 9, что и приводит

к появлению исключения IndexOutOfRangeException, когда происходит обращение

к элементу массива по индексу 4.

Обратите внимание на то, что в операторе catch указан только тип исключения

(в данном случае — IndexOutOfRangeException), а переменная исключения отсутствует.

Как упоминалось ранее, переменную исключения требуется указывать лишь

в том случае, если требуется доступ к объекту исключения. В ряде случаев значение

объекта исключения может быть использовано обработчиком исключений для получения

дополнительной информации о самой ошибке, но зачастую для обработки

исключительной ситуации достаточно просто знать, что она произошла. Поэтому

переменная исключения нередко отсутствует в обработчиках исключений, как в рассматриваемом

здесь примере.

# Второй пример обработки исключительной ситуации

Следует особо подчеркнуть, что весь код, выполняемый в блоке try, контролируется

на предмет исключительных ситуаций, в том числе и тех, которые могут возникнуть

в результате вызова метода из самого блока try. Исключение, генерируемое методом

в блоке try, может быть перехвачено в том же блоке, если, конечно, этого не будет

сделано в самом методе.

В качестве еще одного примера рассмотрим следующую программу, где блок try

помещается в методе Main(). Из этого блока вызывается метод GenException(), в котором

и генерируется исключение IndexOutOfRangeException. Это исключение не

перехватывается методом GenException(). Но поскольку метод GenException() вызывается

из блока try в методе Main(), то исключение перехватывается в блоке catch,

связанном непосредственно с этим блоком try.

(***glava13\_2***)

class ExcTest

{

//generate excpetion

public static void GenException()

{

int[] nums = new int[4];

Console.WriteLine("Befor exception block");

//generate exception index out of range array

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

{

nums[i] = i;

Console.WriteLine("nums [{0}] : {1}", i, nums[i]);

}

Console.WriteLine("No show.");

}

}

class ExcDemo2

{

static void Main()

{

try

{

ExcTest.GenException();

}

catch(IndexOutOfRangeException)

{

//catch except

Console.WriteLine("Index out of range!");

}

Console.WriteLine("After exception block.");

}

}

Как пояснялось выше, метод GenException() вызывается из блока try, и поэтому

генерируемое им исключение перехватывается не в нем, а в блоке catch внутри метода

Main(). А если бы исключение перехватывалось в методе GenException(), оно не

было бы вообще передано обратно методу Main().

# Последствия неперехвата исключений

Перехват одного из стандартных исключений, как в приведенных выше примерах,

дает еще одно преимущество: он исключает аварийное завершение программы. Как

только исключение будет сгенерировано, оно должно быть перехвачено каким-то фрагментом

кода в определенном месте программы. Вообще говоря, если исключение не

перехватывается в программе, то оно будет перехвачено исполняющей системой. Но

дело в том, что исполняющая система выдаст сообщение об ошибке и прервет выполнение

программы.

Такие сообщения об ошибках полезны для отладки программы, но, по меньше

мере, нежелательны при ее использовании на практике! Именно поэтому так важно

организовать обработку исключительных ситуаций в самой программе.

Как упоминалось ранее, тип генерируемого исключения должен соответствовать

типу, указанному в операторе catch. В противном случае исключение не будет перехвачено.

Например, в приведенной ниже программе предпринимается попытка перехватить

ошибку нарушения границ массива в блоке catch, реагирующем на исключение

DivideByZeroException, связанное с делением на нуль и являющееся еще одним

стандартным исключением. Когда индексирование массива выходит за его границы,

генерируется исключение IndexOutOfRangeException, но оно не будет перехвачено

блоком catch, что приведет к аварийному завершению программы.

(***glava13\_3***)

class ExcTypeMismatch

{

static void Main()

{

int[] nums = new int[4];

try

{

Console.WriteLine("Before ecxeption block.");

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

{

nums[i] = i;

Console.WriteLine("nums [{0}] : {1}", i, nums[i]);

}

}

catch(DivideByZeroException)

{

//if catch for DivideByZeroException

//then you cant catch expection

Console.WriteLine("Index out of range!");

}

Console.WriteLine("After exception block.");

}

}

В этом коде catch не работает, потому что в его параметрах указано обработать ошибку деления на нуль, а не выход за пределы массива.

Как следует из приведенного выше результата, в блоке catch, реагирующем

на исключение DivideByZeroException, не удалось перехватить исключение

IndexOutOfRangeException.

# Обработка исключительных ситуаций - “изящный”

# способ устранения программных ошибок

Одно из главных преимуществ обработки исключительных ситуаций заключается

в том, что она позволяет вовремя отреагировать на ошибку в программе и затем продолжить

ее выполнение. В качестве примера рассмотрим еще одну программу, в которой

элементы одного массива делятся на элементы другого. Если при этом происходит

деление на нуль, то генерируется исключение DivideByZeroException. Обработка

подобной исключительной ситуации заключается в том, что программа уведомляет

об ошибке и затем продолжает свое выполнение. Таким образом, попытка деления

на нуль не приведет к аварийному завершению программы из-за ошибки при ее выполнении.

Вместо этого ошибка обрабатывается "изящно", не прерывая выполнение

программы.

(***glava13\_4***)

class ExcDemo3

{

static void Main()

{

int[] numer = { 4, 8, 16, 32, 64, 128 };

int[] denom = { 2, 0, 4, 4, 0, 8 };

for (int i = 0; i < numer.Length; i++)

{

try

{

Console.WriteLine(numer[i] + " / "

+ denom[i] + " = "

+ numer[i] / denom[i]);

}

catch(DivideByZeroException)

{

//catch exception

Console.WriteLine("Not allowed to divade by zero!");

}

}

}

}

Из данного примера следует еще один важный вывод: как только исключение обработано,

оно удаляется из системы. Поэтому в приведенной выше программе проверка

ошибок в блоке try начинается снова на каждом шаге цикла for, при условии, что все

предыдущие исключительные ситуации были обработаны. Это позволяет обрабатывать

в программе повторяющиеся ошибки.

# Применение нескольких операторов catch

С одним оператором try можно связать несколько операторов catch. И на практике

это делается довольно часто. Но все операторы catch должны перехватывать исключения

разного типа. В качестве примера ниже приведена программа, в которой

перехватываются ошибки выхода за границы массива и деления на нуль.

(***glava13\_5***)

class ExcDemo4

{

static void Main()

{

//numer array more than denom

int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };

int[] denom = { 2, 0, 4, 4, 0, 8 };

for(int i = 0; i< numer.Length; i++)

{

try

{

Console.WriteLine(numer[i] + " / "

+ denom[i] + " = "

+ numer[i] / denom[i]);

}

catch(DivideByZeroException)

{

Console.WriteLine("Not Allowed to dived by Zero");

}

catch(IndexOutOfRangeException)

{

Console.WriteLine("Element has not been found.");

}

}

}

}

Как следует из приведенного выше результата, каждый оператор catch реагирует

только на свой тип исключения.

Вообще говоря, операторы catch выполняются по порядку их следования в программе.

Но при этом выполняется только один блок catch, в котором тип исключения

совпадает с типом генерируемого исключения. А все остальные блоки catch

пропускаются.

# Перехват всех исключений

Время от времени возникает потребность в перехвате всех исключений независимо

от их типа. Для этой цели служит оператор catch, в котором тип и переменная исключения

не указываются. Ниже приведена общая форма такого оператора.

catch {

// обработка исключений

}

С помощью такой формы создается "универсальный" обработчик всех исключений,

перехватываемых в программе.

Ниже приведен пример такого "универсального" обработчика исключений. Обратите

внимание на то, что он перехватывает и обрабатывает оба исключения,

IndexOutOfRangeException и DivideByZeroException, генерируемых в программе.

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

class ExcDemo4

{

static void Main()

{

//numer array more than denom

int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };

int[] denom = { 2, 0, 4, 4, 0, 8 };

for (int i = 0; i < numer.Length; i++)

{

try

{

Console.WriteLine(numer[i] + " / "

+ denom[i] + " = "

+ numer[i] / denom[i]);

}

catch

{

Console.WriteLine("Something wrong.");

}

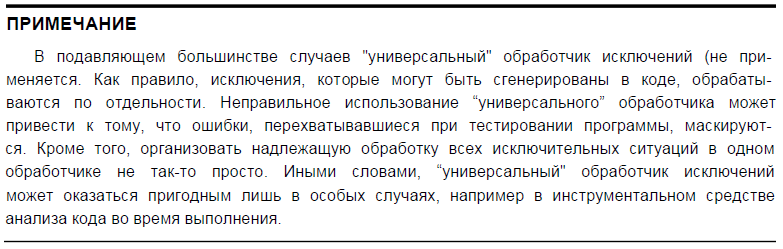
}

}

}

Применяя *"*универсальный*"* перехват, следует иметь в виду, что его блок должен

располагаться последним по порядку среди всех блоков catch.



# Вложение блоков try

Один блок try может быть вложен в другой. Исключение, генерируемое во внутреннем

блоке try и не перехваченное в соответствующем блоке catch, передается во

внешний блок try. В качестве примера ниже приведена программа, в которой исключение

IndexOutOfRangeException перехватывается не во внутреннем, а во внешнем

блоке try.

(***glava15\_6***)

class ExcDemo4

{

static void Main()

{

//numer array more than denom

int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };

int[] denom = { 2, 0, 4, 4, 0, 8 };

try

{

for (int i = 0; i < numer.Length; i++)

{

try

{

Console.WriteLine(numer[i] + " / "

+ denom[i] + " = "

+ numer[i] / denom[i]);

}

catch(DivideByZeroException)

{

Console.WriteLine("Not Allowed to Divide by Zero!");

}

}

}

catch(IndexOutOfRangeException)

{

Console.WriteLine("Out of range!");

}

}

}

В данном примере исключение, обрабатываемое во внутреннем блоке try и связанное

с ошибкой из-за деления на нуль, не мешает дальнейшему выполнению программы.

Но ошибка нарушения границ массива, обнаруживаемая во внешнем блоке try,

приводит к прерыванию программы.

Безусловно, приведенный выше пример демонстрирует далеко не единственное

основание для применения вложенных блоков try, тем не менее из него можно сделать

важный общий вывод. Вложенные блоки try нередко применяются для обработки различных

категорий ошибок разными способами. В частности, одни ошибки считаются

неисправимыми и не подлежат исправлению, а другие ошибки незначительны и могут

быть обработаны немедленно. Как правило, внешний блок try служит для обнаружениям

обработки самых серьезных ошибок, а во внутренних блоках try обрабатываются

менее серьезные ошибки. Кроме того, внешний блок try может стать *"*универсальным*"*

для тех ошибок, которые не подлежат обработке во внутреннем блоке.

# Генерирование исключений вручную throw

В приведенных выше примерах перехватывались исключения, генерировавшиеся

исполняющей системой автоматически. Но исключение может быть сгенерировано

и вручную с помощью оператора throw. Ниже приведена общая форма такого генерирования:

throw exceptOb;

Ниже приведен пример программы, в которой демонстрируется применение оператора

throw для генерирования исключения DivideByZeroException.

(***glava15\_7***)

class ThrowDemo

{

static void Main()

{

try

{

Console.WriteLine("Before except block.");

throw new DivideByZeroException();

}

catch (DivideByZeroException)

{

Console.WriteLine("Exception catched.");

}

Console.WriteLine("After try/catch.");

}

}

Обратите внимание на то, что исключение DivideByZeroException было сгенерировано

с использованием ключевого слова ***new*** в операторе throw. Не следует

забывать, что в данном случае генерируется конкретный объект, а следовательно, он

должен быть создан перед генерированием исключения. Это означает, что сгенерировать

исключение только по его типу нельзя. В данном примере для создания объекта

DivideByZeroException был автоматически вызван конструктор, используемый по

умолчанию, хотя для генерирования исключений доступны и другие конструкторы.

# Повторное генерирование исключений

Исключение, перехваченное в одном блоке catch, может быть повторно сгенерировано

в другом блоке, чтобы быть перехваченным во внешнем блоке catch. Наиболее

вероятной причиной для повторного генерирования исключения служит предоставление

доступа к исключению нескольким обработчикам. Допустим, что один обработчик

оперирует каким-нибудь одним аспектом исключения, а другой обработчик — другим

его аспектом. Для повторного генерирования исключения достаточно указать оператор

throw без сопутствующего выражения, как в приведенной ниже форме.

throw;

Не следует, однако, забывать, что когда исключение генерируется повторно, то оно

не перехватывается снова тем же самым блоком catch, а передается во внешний блок

catch.

В приведенном ниже примере программы демонстрируется повторное

генерирование исключения. В данном случае генерируется исключение

IndexOutOfRangeException.

(***glava15\_7.1***)

class Rethrow

{

public static void GenException()

{

int[] numer = {4,8,16,32,64,128,256,512 };

int[] denom = { 2,0,4,4,0,8};

for(int i = 0; i<numer.Length; i++)

{

try

{

Console.WriteLine(numer[i] + " / "

+ denom[i] + " = "

+ numer[i] / denom[i]);

}

catch (DivideByZeroException)

{

Console.WriteLine("Not allowed to dived by zero.");

}

catch (IndexOutOfRangeException)

{

Console.WriteLine("Out of range");

throw;

}

}

}

}

class RethrowDemo

{

static void Main()

{

try

{

Rethrow.GenException();

}

catch(IndexOutOfRangeException)

{

//catch exception again

Console.WriteLine("Error - program has been stoped.");

}

}

}

В этом примере, как только обработчик получает ошибку из

catch (IndexOutOfRangeException)

{

Console.WriteLine("Out of range");

throw;

}

Выдается сразу сообщение, а потом метод ***throw;*** отправляет обработчику в ***Мain()***

static void Main()

{

try

{

Rethrow.GenException();

}

catch(IndexOutOfRangeException)

{

//catch exception again

Console.WriteLine("Error - program has been stoped.");

}

}

И останавливает программу.

В этом примере программы ошибки из-за деления на нуль обрабатываются локально

в методе GenException(), но ошибка выхода за границы массива генерируется

повторно. В данном случае исключение IndexOutOfRangeException обрабатывается

в методе Main().

# Использование блока finally

Иногда требуется определить кодовый блок, который будет выполняться после выхода

из блока try/catch. В частности, исключительная ситуация может возникнуть

в связи с ошибкой, приводящей к преждевременному возврату из текущего метода.

Но в этом методе мог быть открыт файл, который нужно закрыть, или же установлено

сетевое соединение, требующее разрывания. Подобные ситуации нередки в программировании,

и поэтому для их разрешения в C# предусмотрен удобный способ: воспользоваться

блоком finally.

Для того чтобы указать кодовый блок, который должен выполняться после блока

try/catch, достаточно вставить блок finally в конце последовательности операторов

try/catch. Ниже приведена общая форма совместного использования блоков try/

catch и finally.

try {

// Блок кода, предназначенный для обработки ошибок.

}

catch (ExcepType1 exOb) {

// Обработчик исключения типа ExcepType1.

}

catch (ExcepType2 ехОb) {

// Обработчик исключения типа ЕхсерТуре2.

}

finally {

// Код завершения обработки исключений.

}

Блок finally будет выполняться всякий раз, когда происходит выход из блока try/

catch, независимо от причин, которые к этому привели. Это означает, что если блок

try завершается нормально или по причине исключения, то последним выполняется

код, определяемый в блоке finally. Блок finally выполняется и в том случае, если

любой код в блоке try или в связанных с ним блоках catch приводит к возврату из

метода.

(***glava15\_8***)

class UseFinally

{

public static void GenException(int what)

{

int t;

int[] nums = new int[2];

Console.WriteLine("Get " + what);

try

{

switch (what)

{

case 0: t = 10 / what; break; //generate error Divided by zero

case 1: nums[4] = 4; break; //generate error Index out of range

case 2: return; //return from try block

}

}

catch(DivideByZeroException)

{

Console.WriteLine("Not allowed to divide by zero.");

return; //return from catch block

}

catch(IndexOutOfRangeException)

{

Console.WriteLine("Out of array range.");

}

finally

{

Console.WriteLine("After exit from try block.");

}

}

}

class FinallyDemo

{

static void Main()

{

for(int i = 0; i<3; i++)

{

UseFinally.GenException(i);

Console.WriteLine();

}

}

}

Блок ***finally*** ***выполняется каждый раз*** после того как обработает блок ***catch*** или ***try***.

Как следует из приведенного выше результата, блок finally выполняется независимо

от причины выхода из блока try.

И еще одно замечание: с точки зрения синтаксиса блок finally следует после блока

try, и формально блоки catch для этого не требуются. Следовательно, блок finally

можно ввести непосредственно после блока try, опустив блоки catch. В этом случае

блок finally начнет выполняться сразу же после выхода из блока try, но исключения

обрабатываться не будут.

# Подробное рассмотрение класса Exception

В приведенных выше примерах исключения только перехватывались, но никакой

существенной обработке они не подвергались. Как пояснялось выше, в операторе

catch допускается указывать тип *и* переменную исключения. Переменная получает

ссылку на объект исключения. Во всех исключениях поддерживаются члены, определенные

в классе Exception, поскольку все исключения являются производными от

этого класса.

В классе Exception определяется ряд свойств. К числу самых интересных относятся

три свойства: Message, StackTrace и TargetSite. Все эти свойства доступны

только для чтения. Свойство Message содержит символьную строку, описывающую

характер ошибки; свойство StackTrace — строку с вызовами стека, приведшими к исключительной

ситуации, а свойство TargetSite получает объект, обозначающий метод,

сгенерировавший исключение.

Кроме того, в классе Exception определяется ряд методов. Чаще всего приходится

пользоваться методом ToString(), возвращающим символьную строку с описанием

исключения. Этот метод автоматически вызывается, например, при отображении исключения

с помощью метода WriteLine().

Применение всех трех упомянутых выше свойств и метода из класса Exception

демонстрируется в приведенном ниже примере программы.

(***glava15\_9***)

class ExcTest

{

public static void GenException()

{

int[] nums = new int[4];

Console.WriteLine("Before exception block.");

//out of range

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

{

nums[i] = i;

Console.WriteLine("nums[{0}] : {1}", i, nums[i]);

}

Console.WriteLine("Wount show.");

}

}

class UseExcept

{

static void Main()

{

try

{

ExcTest.GenException();

}

catch(IndexOutOfRangeException exc) //!!!!!

{

Console.WriteLine("Standart message is: ");

Console.WriteLine(exc);

Console.WriteLine("Option StackTrace: " + exc.StackTrace);

Console.WriteLine("Option Message: " + exc.Message);

Console.WriteLine("Option TargetSite: " + exc.TargetSite);

}

Console.WriteLine("After exception block.");

}

}

Первый конструктор используется по умолчанию. Во втором конструкторе указывается

строка *сообщение,* связанная со свойством Message, которое имеет отношение

к генерируемому исключению. В третьем конструкторе указывается так называемое

*внутреннее исключение.* Этот конструктор используется в том случае, когда

одно исключение порождает другое, причем *внутреннее\_исключение* обозначает

первое исключение, которое будет пустым, если внутреннее исключение отсутствует.

(Если внутреннее исключение присутствует, то оно может быть получено из свойства

InnerException, определяемого в классе Exception.) И последний конструктор обрабатывает

исключения, происходящие дистанционно, и поэтому требует десериализации.

Следует также заметить, что в четвертом конструкторе класса Exception типы

SerializationInfo и StreamingContext относятся к пространству имен System.

Runtime.Serialization.

# Наиболее часто используемые исключения

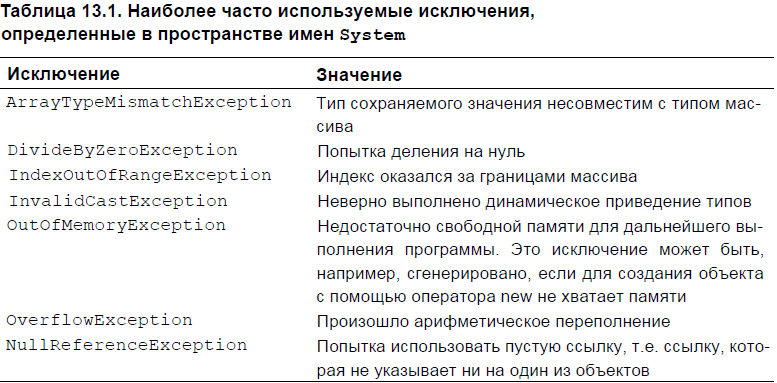
В пространстве имен System определено несколько стандартных, встроенных исключений.

Все эти исключения являются производными от класса SystemException,

поскольку они генерируются системой CLR при появлении ошибки во время выполнения.

В табл. 13.1 перечислены некоторые наиболее часто используемые стандартные

исключения.



Ниже приведен пример программы, демонстрирующий обработку исключения NullReferenceException.

(***glava15\_10***)

class X

{

int x;

public X(int a) { x = a; }

public int Add(X o) { return x + o.x; }

}

//NullReferenceException

class NREDemo

{

static void Main()

{

X p = new X(10);

X q = null; //empty link

int val;

try

{

val = p.Add(q); //Error here

}

catch (NullReferenceException)

{

Console.WriteLine("Exception NullReferenceException!");

Console.WriteLine("Edit error...\n");

//And now fix the problem

q = new X(9);

val = p.Add(q);

}

Console.WriteLine("Value of val = {0}", val);

}

}

В приведенном выше примере программы создается класс X, в котором определяются

член х и метод Add(), складывающий значение члена х в вызывающем объекте

со значением члена х в объекте, передаваемом этому методу в качестве параметра. Оба

объекта класса X создаются в методе Main(). Первый из них (переменная р) инициализируется,

а второй (переменная q) — нет. Вместо этого, переменной q присваивается

пустое значение. Затем вызывается метод р.Add() с переменной q в качестве аргумента.

Но поскольку переменная q не ссылается ни на один из объектов, то при попытке

получить значение члена q.х генерируется исключение NullReferenceException.

# Получение производных классов исключений

В действительности одна из сильных сторон принятого

в C# подхода к обработке исключительных ситуаций состоит в том, что в этом языке

допускается использовать исключения, определяемые пользователем, т.е. тем, кто программирует

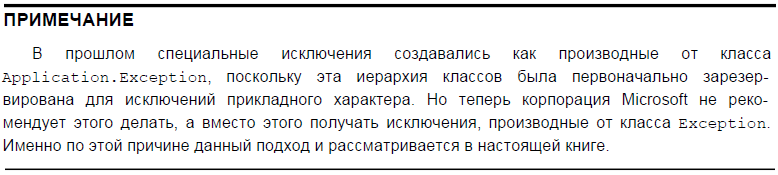
на С#. В частности, такие специальные исключения можно использовать

для обработки ошибок в собственном коде, а создаются они очень просто. Для этого

достаточно определить класс, производный от класса Exception. В таких классах совсем

не обязательно что-то реализовывать — одного только их существования в системе

типов уже достаточно, чтобы использовать их в качестве исключений.



Создаваемые пользователем классы будут автоматически получать свойства и методы,

определенные в классе Exception и доступные для них. Разумеется, любой из этих

членов класса Exception можно переопределить в создаваемых классах исключений.

Когда создается собственный класс исключений, то, как правило, желательно, чтобы

в нем поддерживались все конструкторы, определенные в классе Exception. В простых

специальных классах исключений этого нетрудно добиться, поскольку для этого

достаточно передать подходящие аргументы соответствующему конструктору класса

Exception, используя ключевое слово base. Но формально нужно предоставить только

те конструкторы, которые фактически используются в программе.

В приведенном ниже улучшенном варианте класса RangeArray обработка ошибок нарушения границ массива выполняется более изящным и надежным способом с помощью специально

генерируемого исключения.

(***glava15\_11***)

//Create exception for class RangeArray

class RangeArrayException : Exception

{

//Realize all constructors of Exception class. This constructor

//simply realize constructors of the base class. Since class

//RangeArrayException doesnt add anything to class Exception, then

//dont need any operations

public RangeArrayException() : base() { }

public RangeArrayException(string str) : base(str) { }

public RangeArrayException(string str,

Exception inner) : base(str, inner) { }

protected RangeArrayException(

System.Runtime.Serialization.SerializationInfo si,

System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }

//override ToString() method

public override string ToString()

{

return Message;

}

}

//better variant of RangeArray class

class RangeArray

{

//closed data

int[] a; //base arr link

int lowerBound;

int upperBound;

//auto realize and only for reading Length option

public int Length { get; private set; }

//build array using size

public RangeArray(int low, int high)

{

high++;

if (high <= low)

{

throw new RangeArrayException("Lower index is not less than High");

}

a = new int[high - low];

Length = high - low;

lowerBound = low;

upperBound = --high;

}

//indexator for the class

public int this[int index]

{

get

{

if (ok(index))

{

return a[index - lowerBound];

}

else

{

throw new RangeArrayException("Out of range error.");

}

}

set

{

if (ok(index))

{

a[index - lowerBound] = value;

}

else

{

throw new RangeArrayException("Out of range error.");

}

}

}

//true if index in range

private bool ok(int index)

{

if (index >= lowerBound & index <= upperBound) return true;

return false;

}

}

//demonstrate

class RangeArrayDemo

{

static void Main()

{

try

{

RangeArray ra = new RangeArray(-5, 5);

RangeArray ra2 = new RangeArray(1, 10);

//use ra object as array

Console.WriteLine("Array ra length: " + ra.Length);

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

ra[i] = i;

Console.Write("Contanings of ra: ");

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

Console.Write(ra[i] + " ");

Console.WriteLine("\n");

//use ra2 object as array

Console.WriteLine("Array ra2 length: " + ra2.Length);

for (int i = 1; i <= 10; i++)

ra2[i] = i;

Console.Write("Contanings of ra2: ");

for (int i = 1; i <= 10; i++)

Console.Write(ra2[i] + " ");

Console.WriteLine("\n");

}

catch(RangeArrayException exc)

{

Console.WriteLine(exc);

}

//And now show handling some erros

Console.WriteLine("Generate error of OutOfRange.");

//use wrong indexex int constructor

try

{

RangeArray ra3 = new RangeArray(-2, 2);

for (int i = -2; i <= 2; i++)

ra3[i] = i;

Console.WriteLine("Contanings of ra3: ");

for (int i = -2; i <= 10; i++) //out of range error

Console.Write(ra3[i] + " ");

}

catch (RangeArrayException exc)

{

Console.WriteLine(exc);

}

}

}

Когда возникает ошибка, оператор

throw new RangeArrayException("Out of range error.");

создает объект этого типа с сообщением в скобках. Потом отправляет его в

catch (RangeArrayException exc)

{

Console.WriteLine(exc);

}

Теперь этот объект имеет имя exc, и в консоли выводится заданное ему сообщение заранее.

Когда возникает ошибка нарушения границ массива класса RangeArray, генерируется

объект типа RangeArrayException. В классе RangeArray это может произойти

в трех следующих местах: в аксессоре get индексатора, в аксессоре set индексатора

и в конструкторе класса RangeArray. Для перехвата этих исключений подразумевается,

что объекты типа RangeArray должны быть сконструированы и доступны из

блока try, что и продемонстрировано в приведенной выше программе. Используя

специальное исключение для сообщения об ошибках, класс RangeArray теперь действует

как один из встроенных в C# типов данных, и поэтому он может быть полностью

интегрирован в механизм обработки ошибок, обнаруживаемых в программе.

Обратите внимание на то, что в теле конструкторов класса исключения

RangeArrayException отсутствуют какие-либо операторы, но вместо этого они просто

передают свои аргументы классу Exception, используя ключевое слово base. Как

пояснялось ранее, в тех случаях, когда производный класс исключений не дополняет

функции базового класса, весь процесс создания исключений можно поручить конструкторам

класса Exception. Ведь производный класс исключений совсем не обязательно

должен чем-то дополнять функции, наследуемые от класса Exception.

# Перехват исключений производных классов

При попытке перехватить типы исключений, относящихся как к базовым, так

и к производным классам, следует особенно внимательно соблюдать порядок следования

операторов catch, поскольку перехват исключения базового класса будет совпадать

с перехватом исключений любых его производных классов. Например, класс

Exception является базовым для всех исключений, и поэтому вместе с исключением

типа Exception могут быть перехвачены и все остальные исключения производных от

него классов.

Если требуется перехватывать исключения базового и производного классов, то

первым по порядку должен следовать оператор catch, перехватывающий исключение

производного класса. Это правило необходимо соблюдать потому, что при перехвате

исключения базового класса будут также перехвачены исключения всех производных

от него классов. Правда, это правило соблюдается автоматически: если первым расположить

в коде оператор catch, перехватывающий исключение базового класса, то

во время компиляции этого кода будет выдано сообщение об ошибке.

(***glava15\_12***)

class ExceptA : Exception

{

public ExceptA(string str) : base(str) { }

public override string ToString()

{

return Message;

}

}

//create exc class, derived from ExceptA

class ExceptB : ExceptA

{

public ExceptB(string str) :base(str) { }

public override string ToString()

{

return base.ToString();

}

}

class OrderMatters

{

static void Main()

{

for(int x = 0; x < 3; x++)

{

try

{

if (x == 0) throw new ExceptA("Catch exception type ExceptA");

else if (x == 1) throw new ExceptB("Catch exception type ExceptB");

else throw new Exception();

}

catch (ExceptB exc)

{

Console.WriteLine(exc);

}

catch (ExceptA exc)

{

Console.WriteLine(exc);

}

catch (Exception exc)

{

Console.WriteLine(exc);

}

}

}

}

Обратите внимание на порядок следования операторов catch. Именно в таком порядке

они и должны выполняться. Класс ExceptB является производным от класса

ExceptA, поэтому исключение типа ExceptB должно перехватываться до исключения

типа ExceptA. Аналогично, исключение типа Exception (т.е. базового класса для всех

исключений) должно перехватываться последним. Для того чтобы убедиться в этом,

измените порядок следования операторов catch. В итоге это приведет к ошибке во

время компиляции.