Лабораторная работа №6

Обработка исключительных ситуаций

Цель: овладеть возможностью использования методов и генерации исключений.

Ход работы:

1. Написать программу, в которой использовать обработку следующих исключений: OverflowException, FormatException, сгенерировать свое исключение (UseException).

Теоретические сведения.

Обработка исключений

В среде выполнения поддерживается модель обработки исключений, основанная на понятиях объектов исключения и защищенных блоков кода. Следует отметить, что схеме 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, на которую метод, в котором она произошла, реагирует 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 писать код, обеспечивающий корректный перехват исключений, можно создать собственные классы исключений, получив производные классы из соответствующего базового исключения.

Все классы, описывающие исключения, являются производными от класса **Exception**, хотя напрямую от этого класса наследуются только два – **ApplicationException** и **SystemException**.

Перехват исключений осуществляется при помощи блока **try…catch**, синтаксис которого представлен ниже:

```
        try
{
// код программы, который может бросить исключение
…
}
catch (“класс, порожденный от Exception ” “имя”)
{
// код, выполняющийся в случае возникновения исключения
…
```

```
}
finally
{
// код завершения, обработки исключения
}
```

Эти блоки используются следующим образом:

• try содержит код, который может привести к возникновению исключительной ситуации.

• catch содержит код, который должен выполняться в случае возникновения исключительной ситуации. Блоки catch могут быть настроены таким образом, чтобы реагировать только на один конкретный тип исключительной ситуации (например, на System. IndexOutOfRangeException), поскольку существует возможность одновременно установить несколько блоков catch. Но можно и вообще опустить этот параметр, тогда получится общий блок catch, который будет реагировать на все виды исключительных ситуаций.

• finally содержит блок кода, который выполняется всегда: либо после выполнения блока try, если не возникло никаких исключительных ситуаций, либо после блока catch, если производилась обработка исключительной ситуации, либо непосредственно перед тем, как приложение будет завершено из-за возникновения необрабатываемой исключительной.

Например,

```
try {
        Console.WriteLine("Enter a number");
        int i = int.Parse(Console.ReadLine());
}
catch (OverflowException caught)
{
        Console.WriteLine(caught);
}
```

Существует также возможность использовать блоки try и finally без блока catch, а также блок try с несколькими блоками catch. Если имеется один или несколько блоков catch, то блок finally является необязательным, в противном случае он обязателен.

События, происходящие после возникновения исключительной ситуации в блоке кода try, имеют такую последовательность:

• Выполнение блока try прекращается в той точке, в которой возникла исключительная ситуация.

• Если существует блок catch, то производится проверка на предмет того, совпадает ли его параметр с типом возникшей исключительной ситуации. Если блок catch отсутствует, то выполняется блок finally (который в этом случае является обязательным).

• Если блок catch существует, но значение его параметра не совпадает с типом возникшей исключительной ситуации, проверка производится в других блоках catch.

• Если параметр блока catch совпадает с типом возникшей исключительной ситуации, то сначала выполняется код, содержащийся в блоке, а затем — в случае, если он присутствует,—ВЫПОЛНЯЕТСЯ блок finally.

• Если ни один из блоков catch не соответствует типу возникшей исключительной ситуации, то выполняется блок finally (если он существует).

Просмотр списков и настройка исключительных ситуаций

.NET Framework содержит огромное количество типов исключительных ситуаций, и мы обладаем полной свободой создавать и обрабатывать эти исключительные ситуации в своих программах и даже создавать исключительные ситуации для обработки более сложными приложениями. VS предоставляет специальный диалог, позволяющий просматривать и редактировать имеющиеся исключительные ситуации, который можно вызвать через пункт меню Debug | Exceptions... (или нажатием комбинации клавиш Ctrl+Alt+E).

Исключительные ситуации выводятся в соответствии со своей категорией и пространством имен библиотеки .NET.

Любая исключительная ситуация может быть настроена с помощью переключателей, расположенных в нижней части окна Exceptions. Большинство исключительных ситуаций по умолчанию имеют установку Use parent setting (использовать установку более высокого уровня); это означает, что для них применяются опции уровня всей категории (они полностью совпадают с изображенными на вышеприведенном рисунке). Первая группа переключателей, When the exception is thrown (когда создалась исключительная ситуация), может использоваться для перехода в отладчик даже в том случае, если исключительная ситуация уже обработана. Вторая группа позволяет игнорировать необрабатываемые исключительные ситуации независимо от того, к каким последствиям это может привести.

В большинстве случаев установки по умолчанию нас вполне устраивают.

Наиболее распространенные исключения:

ArrayTypeMismatchException	Тип сохраняемого значения несовместим с типом массива
DivideByZeroException	Попытка деления на нуль
IndexOutOfRangeException	Индекс массива вне диапазона
InvalidCastException	Неверно выполнено динамическое приведение типов
OutOfMemoryException	Обращение к оператору new оказалось неудачным из-за недостаточного объема свободной памяти
OverflowException	Имеет место арифметическое переполнение
RankArrayException	нарушения границ диапазона RangeArray
StackOverflowException	Переполнение стека
NullReferenceException	Была сделана попытка использовать нулевую ссылку, т.е. ссылку, которая не указывает ни на какой объект

Обработка исключений

Можно сгенерировать исключение вручную, используя инструкцию **throw**. Формат ее записан таков:

throw *exceptOb*;

Элемент ***exceptOb*** — это объект класса исключений, производного от класса Exception.

Чаще всего генерируемые исключения являются экземплярами классов исключений, создаваемых в программе.

Например,

```
…
if(high <= low)
 {
throw new RangeArrayException("Нижний индекс не меньше верхнего");
}
…
catch (RangeArrayException exc)
{
Console.WriteLine(exc);
}
```

Методы

В C# методы определяются в рамках объявления класса.

Методы (функции) являются членами класса и определяют:

- функциональность объектов–членов класса (нестатические методы объявляются без спецификатора – методы объектов;

- непосредственно функциональность самого класса (статические методы со спецификатором static – методы класса).

Метод может быть объявлен (метод объявляется в классе) и метод может быть вызван (выражение вызова метода располагается в теле метода).

Методы класса – это не просто процедуры, это процедуры, обслуживающие данные. Все поля доступны любому методу по определению. Нужно четко понимать, что в момент выполнения программной системы работа идет не с классом, а с объектами – экземплярами класса. Из полей соответствующего объекта – цели вызова – извлекается информация, нужная методу в момент вызова, а работа метода чаще всего сводится к обновлению значений полей этого объекта. Поэтому понятно, что методу не нужно через входные аргументы передавать информацию, содержащуюся в полях. Если в результате работы метода обновляется значение некоторого поля, то, опять-таки не нужен никакой выходной аргумент.

Описание методов (процедур и функций). Синтаксис.

Синтаксически в описании метода различают две части – **описание заголовка** и **описание тела** метода:

заголовок_метода

{

тело_метода

}

Рассмотрим синтаксис заголовка метода:

[атрибуты][модификаторы]{void| тип_результата_функции} имя_метода([список_формальных_аргументов])

Имя метода и список формальных аргументов составляют **сигнатуру метода**. Заметьте, в сигнатуру не входят имена формальных аргументов, здесь важны типы аргументов. В сигнатуру не входит и тип возвращаемого результата.

Модификаторы уровня доступа к методу определяют откуда можно будет обратиться к данному методу.

Модификатор	**Назначение**
public	**метод открыт** и доступен для вызова клиентами и потомками класса
private	метод предназначен для внутреннего использования в классе и доступен для вызова только в теле методов самого класса
protected	метод будет доступен как из класса, в котором он определен, так и из любого производного класса. Для остальных вызовов из вне этот метод будет недоступен.
internal	метод будет доступен из всех классов внутри сборки, в которой он определен. Из-за пределов этой сборки обратиться к нему будет нельзя.
protected internal	действует как protected или как internal

```
// Уровни доступа к методам
class SomeClass
(
// Доступен отовсюду
public void MethodA(){};

// Доступен только из типов данных SomeClass
private void MethodB(){};

// Доступен только из SomeClass и из классов, производных от SomeClass
// Сна любом нижестоящем уровне иерархии)
protected void MethodC(){};

// Доступен только из той же самой сборки
internal void MethodD(){};

// Будет действовать как protected или internal
protected internal void MethodE(){};
```

```
// Будет считаться protected - по умолчанию
void MethodO {};
}
```

Обязательным при описании заголовка является указание типа результата, имени метода и круглых скобок, наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует. Формально тип результата метода указывается всегда, но значение **void** однозначно определяет, что метод реализуется процедурой, т.е. не возвращает никакого результата и не использует **return.** Тип результата, отличный от void, указывает на функцию.

Вот несколько простейших примеров описания методов:

void A() {…};

int B(){…};

public void C(){…};

Методы A и B являются закрытыми, а метод C – открыт. Методы A и C реализованы процедурами, а метод B – функцией, возвращающей целое значение.

Методы могут быть объявлены как статические — с использованием ключевого слова **static**. Это значит, что статический метод (static method) может быть вызван напрямую через уровень класса, без необходимости создавать хотя бы один экземпляр объекта данного класса. По этой причине метод Main() всегда объявляется как static — чтобы этот метод мог начать выполняться еще до создания первого экземпляра класса, в котором он определен.

Список формальных аргументов

Как уже отмечалось, список формальных аргументов метода может быть пустым и это довольно типичная ситуация для методов класса. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Рассмотрим теперь синтаксис объявления формального аргумента:

[ref|out|params] тип_аргумента имя_аргумента

Обязательным является указание типа и имени аргумента. Заметьте, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, классом, структурой, интерфейсом, перечислением, функциональным типом.

Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему **произвольное число фактических аргументов**. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово **params**. Оно задается один раз и указывается только для последнего аргумента списка, объявляемого как массив произвольного типа. При вызове метода этому формальному аргументу соответствует произвольное число фактических аргументов.

Выделяют три группы аргументов метода:

- **входные in** (задаются без ключевого слова, передают информацию методу, их значения в теле метода только читаются),

- **выходные out** (результаты метода, они получают значения в ходе работы метода, следовательно, их не надо инициализировать),

- **обновляемые ref** (выполняют обе функции [in, out], их значения используются в ходе вычислений и обновляются в результате работы метода).

Основные отличия параметров сведены в таблицу:

ref [in]	**ref out**
Параметры, передаваемые по ссылке, обязательно должны быть инициализированы перед передачей вызываемому методу, т. к. передается ссылка на существующий тип	Исходящие параметры перед передачей вызываемому методу инициализировать не обязательно. Причина очевидна — о присвоении значения этому параметру должен позаботиться вызываемый метод.

Если аргумент объявлен как выходной с ключевым словом out, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

Выделяют **два способа передачи параметров**:

- по значению (работа с копией переменной, которая уничтожается по окончанию работы метода);

- по ссылке (работа с самой переменной по ссылке).

О соответствии списков формальных и фактических аргументов

Между списком формальных и списком фактических аргументов должно выполняться определенное соответствие по числу, порядку следования, типу и статусу аргументов. Если в первом списке n формальных аргументов, то фактических аргументов должно быть не меньше n (соответствие по числу). Каждому i-му формальному аргументу (для всех i от 1 до n-1) ставится в соответствие i-й фактический аргумент. Последнему формальному аргументу при условии, что он объявлен с ключевым словом params, ставятся в соответствие все оставшиеся фактические аргументы (соответствие по порядку). Если формальный аргумент объявлен с ключевым словом ref или out, то фактический аргумент должен сопровождаться таким же ключевым словом в точке вызова (соответствие по статусу).

Если формальный аргумент объявлен с типом T, то выражение, задающее фактический аргумент должно быть согласовано по типу с типом T – допускает преобразование к типу T, совпадает с типом T или является его потомком (соответствие по типу).

Если формальный аргумент является выходным – объявлен с ключевым словом ref или out, то соответствующий фактический аргумент не может быть выражением, поскольку используется в левой части оператора присваивания, так что он должен быть именем, которому можно присвоить значение.

Тело метода

Синтаксически тело метода является **блоком**, представляющим последовательность операторов и описаний переменных, заключенную в фигурные

скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор перехода, возвращающий значение функции в форме return(выражение).

Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.

Вызов метода.

Синтаксис.

Как уже отмечалось, метод может вызываться в выражениях или быть вызван как оператор. В качестве оператора может использоваться любой метод – как процедура, так и функция. Конечно, функцию разумно вызывать как оператор только, если она обладает побочным эффектом. В последнем случае она вызывается ради своего побочного эффекта, а возвращаемое значение никак не используется. Подобную роль играет использование некоторых выражений с побочным эффектом в роли оператора, классическим примером является оператор x++;.

Если же попытаться вызвать процедуру в выражении, то это приведет к ошибке еще на этапе компиляции. Возвращаемое процедурой значение void не совместимо с выражениями. Так что в выражениях могут быть вызваны только функции.

Сам **вызов метода**, независимо от того, процедура это или функция, имеет один и тот же синтаксис:

имя_метода([список_фактических_аргументов])

Если это оператор, то вызов завершается точкой с запятой. Формальный аргумент, задаваемый при описании метода, это всегда имя аргумента (идентификатор). Фактический аргумент – это выражение, значительно более сложная синтаксическая конструкция. Вот точный синтаксис фактического аргумента:

[ref|out]выражение

Семантика.

Что происходит в момент вызова метода? Выполнение начинается с вычисления фактических аргументов, которые, как мы знаем, являются выражениями. Вычисление этих выражений может приводить в свою очередь к вызову других методов. Так что это первый этап может быть довольно сложным и требовать больших временных затрат. В чисто функциональном программировании все вычисление по программе сводится к вызову одной функции, фактическими аргументами которой являются вызовы функций и так далее и так далее.

Для простоты понимания семантики вызова можно полагать, что в точке вызова создается блок, соответствующий телу метода (реально все происходит значительно эффективнее). В этом блоке происходит замена имен формальных аргументов фактическими аргументами. Для выходных аргументов, для которых фактические аргументы также являются именами, эта замена или передача аргументов происходит **по ссылке**, означая замену формального аргумента ссылкой на реально существующий объект, заданный фактическим аргументом. Чуть более сложную семантику имеет **вызов по значению**, применяемый к формальным аргументам, объявленным без ключевых слов ref и out. При вычислении выражений, заданных такими фактическими аргументами, их

значения присваиваются специально создаваемым переменным, локализованным в теле исполняемого блока. Имена этих локализованных переменных и подставляются вместо имен формальных аргументов. Понятно, что тип локализованных переменных определяется типом соответствующего формального аргумента. Понятно также, что семантика замены формальных аргументов фактическими – это по сути семантика оператора присваивания.

Каково следствие семантики вызова по значению? Если вы забыли указать ключевое слово ref или out для аргумента, фактически являющегося выходным, то к нему будет применяться вызов по значению. Даже если в теле метода происходит изменение значения этого аргумента, то оно действует только на время выполнения тела метода. Как только метод заканчивает свою работу (завершается блок) все локальные переменные (в том числе созданные для замены формальных аргументов) оканчивают свое существование, так что изменения не затронут фактических аргументов и они сохранят свое значение, бывшее у них до вызова. Отсюда вывод: все выходные аргументы, значения которых предполагается изменить в процессе работы, должны иметь ключевое слово ref или out. Еще один важный вывод: Ключевым словом ref полезно иногда снабжать и входные аргументы. Если известно, что фактический аргумент будет всегда представлен именем, а не сложным выражением, то в целях экономии памяти разумно для таких аргументов применять семантику вызова по ссылке. В этом случае не будет создаваться копия аргумента, это экономит память и время, что может быть важно при работе со сложными структурами.

Говоря о семантике вызова по ссылке и по значению, следует сделать одно важное уточнение. В объектном программировании, каковым является и программирование на C#, основную роль играют ссылочные типы – мы работаем с классами и объектами. Когда методу передается объект ссылочного типа, то все поля этого объекта могут в методе меняться самым беззастенчивым образом. И это несмотря на то, что объект формально не является выходным, не имеет ключевых слов ref или out, использует семантику вызова по значению. Сама ссылка на объект при этом, как и положено, остается неизменной, но состояние объекта, его поля могут полностью обновиться. Такая ситуация типична и представляет один из основных способов изменения состояния объектов. Именно поэтому ref или out не часто появляются при описании аргументов метода.

Перегрузка методов.

Под перегрузкой методов понимается использование в одном классе нескольких методов с одним именем.

Приведем пример (TestMethod), в котором создадим класс StrMethod, где опишем три метода с одним именем, каждый из которых выполняет различные действия:

```
class StrMethod
   {
      static void Str()
      {
         Console.WriteLine("My test ");
      }
```

```
        static void Str(string mystring)
        {
            Console.WriteLine("My test string" + mystring);
        }

        static void Str(string mystr1, string mystr2)
        {
            Console.WriteLine("My test string 1 is"+mystr1);
            Console.WriteLine("My test string 2 is" + mystr2);
        }

    }
```

Эти методы компилятор различает по списку параметров, анализируя их тип и метод передачи.

Если два метода отличаются только возвращаемым значением, то это вызовет ошибку компиляции.

Если два метода отличаются только именами параметров, то для компилятора они идентичны.

Использование перегруженных методов необходимо тогда, когда нам нужно реализовать какую-то функциональность и передавать в метод разное количество параметров. Либо когда необходимо расширить существующую функциональность.

Задания для лабораторной работы

1. Функция вычисляет площадь треугольника по трем сторонам

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где $p = (a + b + c) / 2$.

2. Функция вычисляет корень линейного уравнения $ax + b = 0$.
3. Функция вычисляет периметр треугольника.
4. Функция переводит часы и минуты в секунды.
5. Функция вычисляет корень квадратного уравнения $ax^2 + bx + c = 0$.
6. Функция вычисляет сумму геометрической прогрессии $S_n = (a_0 - a_n r) / (1 - r)$.
7. Функция выполняет деление нечетких чисел A и B. Числа представлены структурой с двумя полями (см. задание 1.31).
8. Функция вычисляет нечеткое число, обратное заданному. Числа представлены структурой с двумя полями (см. задание 1.31).
9. Функция выполняет деление комплексных чисел A и B. Комплексные числа представлены структурой-парой (см. задание 1.21).
10. Функция вычисляет целую часть неправильной дроби, представленной числителем и знаменателем — целыми числами.
11. Функция переводит комплексное число $z = x + i\,y$ из алгебраической формы в тригонометрическую z = radius(cos(angle) + i sin(angle)). Комплексное число z представлено структурой-парой (см. задание 1.21). Преобразованное число тоже представляется структурой-парой (radius, angle):

$$\text{radius} = \sqrt{x^2 + y^2}; \quad \text{angle} = \arcsin\frac{y}{x}$$

12. Функция вычисляет разность между двумя датами в днях. Даты представлены структурой с тремя полями: год, месяц, день.
13. Функция вычисляет продолжительность телефонного разговора в минутах,
14. Функция вычисляет день недели по дате. Даты представлены структурой с тремя полями: год, месяц, день. Первое января считается понедельником.
15. Функция вычисляет углы прямоугольного треугольника. В качестве параметров передаются катеты A и B. (Синус угла A1, противолежащего катету A, вычисляется по формуле sin (A1) = a / c, где c — гипотенуза треугольника.)
16. Функция проверяет, является ли передаваемая строка палиндромом.
17. Функция определяет, существуют ли прямые $A_1x + B_1y + C_1 = 0$ и $A_2x + B_2y + C_2 = 0$, если выражение $d = A_1B_2 - A_2B_1$ не равно нулю. Прямые задаются структурой с тремя полями.

18. Функция вычисляет расстояние между двумя точками $P_1(x_1, y_1)$ и $P_2(x_2, y_2)$ по формуле

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Исключение генерируется, когда P_1 и P_2 — одна и та же точка.

19. Функция вычисляет расстояние от точки $P(x_1, y_1)$ до прямой $Ax + By + C = 0$ по формуле

$$D = \frac{|Ax_1 + By_1 + C|}{\sqrt{A^2 + B^2}}$$

20. Функция выясняет, является ли год високосный. Високосность определяется следующим образом: если номер года не делится на 100, то високосным считается тот, который делится на 4 без остатка; если номер года делится на 100, то номер високосного года делится на 400 без остатка.

Выполнить задания 1–20, реализовав подмену стандартной функции `unexpected()`. Пользовательская функция должна выводить сообщение об отсутствии обработчика исключения и заканчивать работу.