Лекция 21

Механизм обработки и генерирования исключительных ситуаций

Исключительные ситуации

Исключительная ситуация (**ИС**), или просто исключение, - это ошибка, которая происходит во время выполнения программы.

Обработка исключительных ситуаций рационализирует весь процесс обработки ошибок, позволяя определить в программе блок кода, называемый обработчиком исключений и выполняющийся автоматически, когда возникает ошибка.

Перехват исключений исключает аварийное завершение программы. Одно из главных **преимуществ** обработки исключительных ситуаций заключается в том, что она позволяет вовремя отреагировать на ошибку в программе и затем продолжить ее выполнение.

Исключительные ситуации

В классе **Exception** определяются четыре конструктора.

public Exception ()

public Exception(string сообщение)

public Exception(string сообщение, Exception внутреннее_исключение)

protected Exception(System.Runtime.Serialization.SerializationInfo информация,
System.Runtime.Serialization.StreamingContext контекст)

Классы исключений

ArrayTypeMismatchException Тип сохраняемого значения несовместим с типом массива

DivideByZeroException Попытка деления на нуль

indexOutOfRangeException Индекс за границами массива

invalidCastException Неверно выполнено динамическое приведение типов

OutOfMemoryException Недостаточно свободной памяти для дальнейшего выполнения программы. Это исключение может быть, например, сгенерировано, если для создания объекта с помощью оператора new не хватает памяти

OverflowException Арифметическое переполнение

NullReferenceException Попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Обработка исключительных ситуаций

Обработка исключительных ситуаций в **C#** организуется с помощью четырех ключевых слов: **try**, **catch**, **throw** и **finally**.

Обработка исключительных ситуаций

Исключения обнаруживаются и обрабатываются в операторе **try**, который содержит три части:

1. ***контролируемый блок*** — составной оператор, предваряемый ключевым словом **try**. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;

Обработка исключительных ситуаций

2. один или несколько ***обработчиков исключений*** — блоков **catch**, в которых описывается, как обрабатываются ошибки различных типов;

3. ***блок завершения*** **finally** выполняется независимо от того, возникла ошибка в контролируемом блоке или нет.

Синтаксис оператора **try**:

try блок [блоки **catch**] [блок **finally**]

Отсутствовать могут либо блоки **catch**, либо блок **finally**, но не оба одновременно.

Обработка исключительных ситуаций

Порядок обработки исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки в блоке **try**. Функция или операция, в которой возникла ошибка, генерирует исключение.

2. Выполнение текущего блока **try** прекращается, отыскивается соответствующий обработчик исключения **catch** , и ему передается управление.

3. Выполняется блок **finally**, если он присутствует.

4. Если обработчик не найден, вызывается стандартный обработчик исключения. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

Обработка исключительных ситуаций

Обработчики исключений должны располагаться непосредственно за блоком **try**. Они начинаются с ключевого слова **catch**, за которым в скобках следует тип обрабатываемого исключения.
Блоки **catch** просматриваются в том порядке, в котором они записаны, пока не будет найден соответствующий типу выброшенного исключения.

Существуют три формы записи обработчиков:
catch(тип имя) { ... /* тело обработчика */ }
catch(тип) { ... /* тело обработчика */ }
catch { ... /* тело обработчика */ }

Обработка исключительных ситуаций

```
try {
      ... // Блок кода, проверяемый на наличие ошибок
}
catch ( OverflowException e ) {
   ... // Обработка исключений класса
      // OverflowException (переполнение)
}
catch ( DivideByZeroException ) {
   ... // Обработка исключений класса
      // DivideByZeroException (деление на 0)
}
catch {
      ...    // Обработка всех остальных исключений
}
```

Пример 1

```
using System;
class ExcDemo4 {
   static void Main() {
      // Массив numer длиннее массива 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++) {   // нет ошибки
         try {
               Console.WriteLine(numer[i] + " / " + denom[i] +
    " равно " + numer[i]/denom[i]);
         }
```

```
//  for (int i=0; i < 10; i++) {       // ошибка
```

Пример 1

```
        catch (DivideByZeroException) {
          Console.WriteLine("Делить на нуль нельзя!");
        }
        catch (IndexOutOfRangeException) {
          Console.WriteLine("Подходящий элемент не
    найден.");
        }
        catch { // "Универсальный" перехват
          Console.WriteLine("Возникла некоторая
    исключительная ситуация.");
        }
      }
    }
}
```

Обработка исключительных ситуаций

Операторы **try** могут многократно вкладываться друг в друга. Исключение, которое возникло во внутреннем блоке **try** и не было перехвачено соответствующим блоком **catch**, передается на верхний уровень, где продолжается поиск подходящего обработчика. Этот процесс называется *распространением исключения*.

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.

Пример 2

```
using System;
class NestTrys {
   static void Main() {
      int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int[] denom ={2, 0, 4, 4, 0, 8};
      try { // внешний блок try
         for (int i=0; i < numer.Length; i++) {
            try { // вложенный блок try
               Console.WriteLine(numer[i] + " / " + denom[i] + "
   равно " + numer[i]/denom[i]);
            }
```

Пример 2

```
        catch (DivideByZeroException) {
          Console.WriteLine("Делить на нуль нельзя!");
        }
      }
    }
    catch (IndexOutOfRangeException) {
      Console.WriteLine("Подходящий элемент не
  найден.");
      Console.WriteLine("Неисправимая ошибка -
  программа прервана.");
    }
  }
}
```

Пример 2

Выполнение программы приводит к следующему результату.

4/2 равно 2

Делить на нуль нельзя!

16/4 равно 4

32/4 равно 8

Делить на нуль нельзя!

128 / 8 равно 16

Подходящий элемент не найден.

Неисправимая ошибка - программа прервана.

Обработка исключительных ситуаций

Для **генерации исключения** используется оператор **throw** с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного класса **System.Exception**.

throw [выражение];

Форма без параметра применяется только внутри блока **catch** для повторной генерации исключения. Тип выражения, стоящего после **throw**, определяет тип исключения.

При генерации исключения выполнение текущего блока прекращается и происходит поиск соответствующего обработчика с передачей ему управления. Обработчик считается найденным, если тип объекта, указанного после **throw**, либо тот же, что задан в параметре **catch**, либо является производным от него.

Пример 3

```
using System;
class ThrowDemo {
   static void Main() {
      try {
         string message = Console.ReadLine();
         if (message.Length > 6)
            throw new Exception("Длина строки больше 6
    символов");
      }
      catch (Exception e) {
         Console.WriteLine("Ошибка: " + e.Message);
      }
      Console.ReadLine();
  }
}
```

Обработка исключительных ситуаций

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. Для повторного генерирования исключения достаточно указать оператор **throw** без сопутствующего выражения.

throw;

Когда исключение генерируется повторно, то оно не перехватывается снова тем же самым блоком **catch**, а передается во внешний блок **catch**.

Пример 4

```
using System;
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] + "
  равно " + numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        Console.WriteLine("Делить на нуль нельзя!");
      }
```

Пример 4

```
        catch (IndexOutOfRangeException) {
          Console.WriteLine("Подходящий элемент не найден.");
          throw; // сгенерировать исключение повторно
        }
      }
    }
}
class RethrowDemo {
  static void Main() {
    try {    Rethrow.GenException ();
    }
    catch(IndexOutOfRangeException) { // перехватить ИС  повторно
      Console.WriteLine("Неисправимая ошибка - программа
   прервана.");
    }
  }
}
```

Пример 5

Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий.

```
int x = 1;
int y = 0;
try{
    int result = x / y;
}
catch(Exception ex) when (y==0){
    Console.WriteLine("y не должен быть равен 0");
}
catch(Exception ex){
    Console.WriteLine(ex.Message);
}
```

Обработка исключительных ситуаций

```
try {
   // Блок кода, предназначенный для обработки ошибок
}
catch (ExcepType1 exOb) {
   // Обработчик исключения типа ExcepType1
}
catch (ExcepType2 exOb) {
   // Обработчик исключения типа ExcepType2
}
finally {
   // Код завершения обработки исключений
}
```

Пример 6

```
using System;
class UseFinally {
   public static void GenException(int what) {
      int t;
      int[] nums = new int [2];
      Console.WriteLine("Получить " + what);
      try {
         switch(what) {
            case 0: t = 10 / what; // ошибка деления на нуль
                    break;
           case 1: nums[4] = 4; // ошибка индексирования массива
                   break;
           case 2: return; // возврат из блока try
         }
      }
```

Пример 6

```
    catch (DivideByZeroException) {
      Console.WriteLine("Делить на нуль нельзя!");
      return; // возврат из блока catch
    }
    catch (IndexOutOfRangeException) {
      Console.WriteLine("Совпадающий элемент не найден.");
    }
    finally { Console.WriteLine("После выхода из блока try."); }
  }
}
class FinallyDemo {
  static void Main() {
    for (int i=0; i < 3; i++) {  UseFinally.GenException(i);
                                 Console.WriteLine();   }
  }
}
```

Пример 6

Результат выполнения программы.

Получить 0

Делить на нуль нельзя

После выхода из блока try.

Получить 1

Совпадающий элемент не найден.

После выхода из блока try.

Получить 2

После выхода из блока try.

Обработка исключительных ситуаций

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**().

Пример 7

```
using System;
class ExcTest {
   public static void GenException() {
      int[] nums = new int[4];
      Console.WriteLine("До генерирования
    исключения.");
   // Исключение в связи с выходом за границы массива
      for (int i=0; i < 10; i++) {
         nums[i] = i;
         Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }
      Console.WriteLine("Не подлежит выводу");
   }
}
```

Пример 7

```
class UseExcept {
  static void Main() {
    try {
      ExcTest.GenException();
    }
    catch (IndexOutOfRangeException exc) {
      Console.WriteLine("Стандартное сообщение таково: ");
      Console.WriteLine(exc); // вызвать метод ToString()
      Console.WriteLine("Свойство StackTrace: " + exc.StackTrace);
      Console.WriteLine("Свойство Message: " + exc.Message);
      Console.WriteLine("Свойство TargetSite: " + exc.TargetSite);
    }
    Console.WriteLine("После блока перехвата исключения.");
  }
}
```

Пример 7

До генерирования исключения. // Результат
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Стандартное сообщение таково:
    System.IndexOutOfRangeException: Индекс находился вне
    границ массива.
в ExcTest.genException() в <имя_файла>:строка 15
в UseExcept.Main()в <имя_файла>:строка 2 9
Свойство StackTrace: в ExcTest.genException()в <имя_файла>:
    строка 15
в UseExcept.Main()в <имя_файла>:строка 29
Свойство Message: Индекс находился вне границ массива.
Свойство TargetSite: Void genException ()
После блока перехвата исключения.

Пример 8

```
using System;
class X {
   int x;
   public X(int a) { x = a; }
   public int Add(X o) {  return x + o.x; }
}
// Продемонстрировать генерирование и обработку
// исключения NullReferenceException
class NREDemo {
   static void Main() {
      X p = new X(10);
      X q = null; // Присвоить явным образом
                  // пустое значение переменной q
      int val;
```

Пример 8

```
    try {
      val = p.Add(q); // Эта операция приведет
                      // к исключительной ситуации
    } catch (NullReferenceException) {
      Console.WriteLine("Исключение
  NullReferenceException!");
      Console.WriteLine("Исправление ошибки...\n");
      // А теперь исправить ошибку
      q = new X(9);
      val = p.Add(q);
    }
    Console.WriteLine("Значение val равно {0}", val);
  }
}
```

Пример 9

```
using System;
// Создать пользовательское исключение для класса RangeArray
class RangeArrayException : Exception {
public RangeArrayException() : base() { }
   public RangeArrayException(string str) : base(str) { }
   public RangeArrayException (string str, Exception inner) : base(str,
    inner) { }
   protected
    RangeArrayException(System.Runtime.Serialization.SerializationI
    nfo si, System.Runtime.Serialization.StreamingContext sc) :
    base(si, sc) { }
   // Переопределить метод ToString() для класса исключения
   // RangeArrayException
   public override string ToString() {
     return Message;
   }
}
```

Пример 9

```
class RangeArray {
   int[] a; // ссылка на базовый массив
   int lowerBound; // наименьший индекс
   int upperBound; // наибольший индекс
   public int Length { get; private set; }
   public RangeArray(int low, int high) {
       high++;
      if (high <= low) {
          throw new RangeArrayException("Нижний индекс
    не меньше верхнего.");
      }
      a = new int[high - low];       Length = high - low;
      lowerBound = low;       upperBound = --high;
   }
```

Пример 9

```
public int this[int index] {
  get {
    if (ok(index)) { return a[index - lowerBound]; }
    else {
      throw new RangeArrayException("Ошибка
нарушения границ.");
    }
  }
  set {
    if (ok(index)) { a[index - lowerBound] = value; }
    else throw new RangeArrayException("Ошибка
нарушения границ.");
}
```

Пример 9

```
  private bool ok(int index) {
      if (index >= lowerBound & index <= upperBound)
   return true;
      return false;
  }
}

class RangeArrayDemo {
  static void Main() {
    try {
        RangeArray ra = new RangeArray(-5, 5);
        RangeArray ra2 = new RangeArray(1, 10);
        Console.WriteLine("Длина массива ra: " + ra.Length);
```

Пример 9

```
  for (int i = -5; i <= 5; i++)  ra[i] = i;
   Console.Write("Содержимое массива га: ");
  for (int i = -5; i <= 5; i++)  Console.Write(ra[i] + " ");
   Console.WriteLine("\n");
  Console.WriteLine("Длина массива га2: " + ra2.Length);
   for (int i = 1; i <= 10; i++) ra2[i] = i;
   Console.Write("Содержимое массива га2: ");
   for (int i = 1; i <= 10; i++)
          Console.Write(ra2[i] + " ");
   Console.WriteLine("\n");
} catch (RangeArrayException exc) {
   Console.WriteLine(exc);
}
```

Пример 9

```
// Продемонстрировать обработку некоторых ошибок

   Console.WriteLine("Сгенерировать ошибки
  нарушения границ.");
   // Использовать неверно заданный конструктор
   try {
       RangeArray ra3 = new RangeArray(100, -10); // ИС!
   } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
   }
```

Пример 9

```
    // Использовать неверно заданный индекс
    try {
       RangeArray ra3 = new RangeArray(-2, 2);
       for (int i = -2; i <= 2; i++)  ra3[i] = i;
       Console.Write("Содержимое массива ra3: ");
     // Сгенерировать ошибку нарушения границ
       for (int i = -2; i <= 10; i++)
            Console.Write(ra3[i] + " ");
    } catch (RangeArrayException exc) {
         Console.WriteLine(exc);
    }
  }
}
```

Применение checked и unchecked

В **C#** допускается указывать, будет ли в коде сгенерировано исключение при переполнении, с помощью ключевых слов **checked** и **unchecked**. Так, если требуется указать, что выражение будет проверяться на переполнение, следует использовать ключевое слово **checked**, а если требуется проигнорировать переполнение — ключевое слово **unchecked**. В последнем случае результат усекается, чтобы не выйти за пределы диапазона представления чисел для целевого типа выражения.

Применение checked и unchecked

У ключевого слова **checked** имеются две общие формы. В одной форме проверяется конкретное выражение, и поэтому она называется **операторной**. А в другой форме проверяется блок операторов, и поэтому она называется **блочной**. Ниже приведены обе формы:

```
checked (выражение)
checked {
  // проверяемые операторы
}
```

где выражение обозначает проверяемое выражение. Если вычисление проверяемого выражения приводит к переполнению, то генерируется исключение **OverflowException**.

Применение checked и unchecked

У ключевого слова **unchecked** также имеются две общие формы. В первой, **операторной** форме переполнение игнорируется при вычислении конкретного выражения.

А во второй, **блочной** форме оно игнорируется при выполнении блока операторов:

unchecked (выражение)

unchecked {

// операторы, для которых переполнение игнорируется

}

где **выражение** обозначает конкретное выражение, при вычислении которого переполнение игнорируется. Если же в непроверяемом выражении происходит переполнение, то результат его вычисления усекается.

Пример 10

```
namespace ConsoleApplication1{
   class Program {
     static void Main(){
       byte a, b, result;
       Console.Write("Введите количество опросов: ");
       int i = int.Parse(Console.ReadLine());
       for (int j = 1; j <= i; j++) {
          try {
              Console.Write("Введите a: ");
            // unchecked используется в одном выражении
              a = unchecked((byte)int.Parse(Console.ReadLine()));
              Console.Write("Введите b: ");
              b = unchecked((byte)int.Parse(Console.ReadLine()));
```

Пример 10

```
            checked { // используется для всего блока операторов
                result = (byte)(a + b);
                Console.WriteLine("a + b = " + result);
                result = (byte)(a * b);
                Console.WriteLine("a*b = " + result + "\n");
            }
        }
        catch (OverflowException) {
            Console.Write("Переполнение\n\n");
        }
    }
     Console.ReadLine();
  }
 }
}
```

Контрольные вопросы

1. Что такое исключительные ситуации?
2. Каковы принципы обработки исключительных ситуаций?
3. Каким образом можно сгенерировать исключительную ситуацию?
4. Зачем нужно повторное генерирование исключений?
5. Для чего применяются ключевые слова checked и unchecked?