Лекция 11. События и обработка исключений

События и обработчики

Все элементы пользовательского интерфейса наследуются от класса `Control`, который содержит большое количество свойств, событий и методов. Например, класс `Control` реализует (а класс `Form` наследует) событие `Click`, которое инициируется всякий раз, когда пользователь щелкает кнопкой мыши элемент управления (или клиентскую область в форме). В исходном тексте класса `Control` событие `Click` определяется примерно так:

```
public event EventHandler Click;
```

Ключевое слово `event` языка C# определяет этот член как событие. В определении события `Click` есть ссылка на *делегат* (*delegate*) `EventHandler`. В пространстве имён `System` оно определяется так:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Делегат — это прототип функции, диктующий определение обработчика события `Click` программе.

Обработчик события — это метод, который автоматически вызывается при генерации события. Метод-обработчик назначается в программе, использующей экземпляр класса, который содержит событие. Для события можно назначить несколько обработчиков, хотя обычно обработчик у события один.

Обработчику событий можно присвоить любое имя, но у него должно быть то же возвращаемое значение, что и у делегата, и те же два параметра, которые также можно переименовать.

```
void MyClicker(object sender, EventArgs e)
{
}
```

Обработчик можно подключить к событию, указав:

1) объект, реализующий событие;
2) само событие;
3) делегат, связанный с событием;
4) обработчик событий.

У этого выражения особый синтаксис:

```
form.Click += MyClicker;
```

Обратите внимание на составной знак *подключения* (*attach*) обработчика события: `+=`. Знак `-=` в этом выражении позволяет *отключить* (*detach*) обработчик событий, но это редко требуется. И, конечно, метод `MyClicker` должен также определяться как делегат `EventHandler`.

Теперь при каждом щелчке кнопкой мыши в клиентской области формы объект `Form` инициирует событие `Click`, что приводит к вызову метода `MyClicker` в программе. Параметр `sender` — это объект, инициирующий событие. В данном случае это созданный программой объект `Form`. В обработчике событий его можно привести к типу формы:

```
MainForm form = (MainForm)sender;
```

или

```
MainForm form = sender as MainForm;
```

Затем обработчик событий может обращаться к свойствам и методам объекта `Form`. Параметр `e` обычно содержит специфические параметры, но для обработчика события `Click` формы они отсутствуют.

Еще одно важное событие, `Paint`, указывает, когда клиентская область элемента управления или формы требует прорисовки. Первое событие `Paint` инициируется при создании формы, а последующие — при её свёртывании и восстановлении, при изменении размеров или когда она выводится на передний план из-под другой

формы. Обработка события Paint не ограничена только отрисовкой поверхности формы — обычно её также используют для восстановления отображаемых в форме текста или графики.

Обработчик события Paint определён в классе Control следующим образом:

```
public event PaintEventHandler Paint;
```

Делегат PaintEventHandler определён в пространстве имён System.Windows.Forms:

```
public delegate void PaintEventHandler(object sender, PaintEventArgs e);
```

Таким образом, в определении обработчика события Paint второй параметр должен иметь тип PaintEventArgs, а не EventArgs:

```
void MyPainter(object sender, PaintEventArgs e)
{
}
```

Программа устанавливает этот обработчик событий с помощью делегата PaintEventHandler:

```
form.Paint += MyPainter;
```

Класс PaintEventArgs определён в пространстве имён System.Windows.Forms. Он наследуется от класса EventArgs и содержит два свойства — Graphics и ClipRectangle. Обработчик событий использует объект Graphics из-за определённых в нем методов прорисовки графических фигур. Свойство ClipRectangle означает прямоугольник, содержащий область, ограничивающую объект Graphics.

Аналогичным образом можно создать в своём классе собственное событие. Если событию не нужны особые параметры, то можно использовать делегат EventHandler из пространства имён System, который описан выше. В противном случае требуется создать класс, производный от EventArgs, добавить в него необходимые свойства и использовать его в качестве типа второго параметра собственного делегата.

Чтобы вызвать событие, используется код, подобный следующему:

```
public event EventHandler myEvent;

...

if (myEvent != null)
{
    myEvent(this, new EventArgs());
}
```

В языке C# перед вызовом события необходимо проверять, зарегистрирован ли для него хотя бы один обработчик. При попытке вызвать событие без обработчика исполняющая среда сгенерирует исключение. Регистрация обработчика для собственного события выполняется так же, как и для любого другого. Когда событие происходит, все его обработчики вызываются в том же порядке, в котором они были зарегистрированы.

Безопасность и обработка исключений

Среда CLR включает мощную поддержку исключений. Исключения могут создаваться и генерироваться в точке, где выполнение кода не может быть продолжено, поскольку возникли некоторые исключительные условия (обычно сбой метода или некорректное состояние). Написание безопасного в отношении исключений кода — настоящее искусство, которым следует овладеть. Было бы ошибкой предполагать, что единственной задачей во время написания безопасного к исключениям кода является генерация исключения при возникновении ошибки и перехват его в некоторой точке. Такое представление безопасного к исключениям кода недальновидно и является прямым путем к безысходности. На самом деле безопасное к исключениям кодирование означает гарантию целостности системы перед лицом возникающих исключений. Когда исключение генерируется, исполняющая система последовательно «раскручивает» стек, выполняя очистку. Задача программиста —

структурировать код таким образом, чтобы при раскручивании стека целостность состояния объектов не нарушалось. В этом суть приёмов безопасного к исключениям кодирования.

Как только исключение сгенерировано, CLR начинает процесс итеративной «раскрутки» стека исключений. Делая это, она очищает все объекты, локальные по отношению к каждому фрейму стека. В некоторой точке фрейм в стеке может иметь обработчик исключения, зарегистрированный именно для типа сгенерированного исключения. Как только среда CLR достигает этого фрейма, она вызывает этот обработчик, чтобы справиться с ситуацией. Если стек полностью раскручен, а обработчик для сгенерированного исключения не найден, может быть инициировано событие «необработанное исключение» для текущего домена приложения, и выполнение приложения будет прервано.

Генерация исключений

Акт генерации исключения достаточно прост. Нужно просто выполнить оператор `throw`, параметром которого является генерируемое исключение. Например, предположим, что написан собственный класс коллекции, позволяющий пользователям обращаться к элементам по индексу, и необходимо уведомить пользователя, когда в параметре передается неверный индекс. Для этого можно сгенерировать исключение `ArgumentOutOfRange`, как показано в следующем коде:

```
public class MyCollection
{
    private int count;

    public object GetItem(int index)
    {
        if (index < 0 || index >= count)
        {
            throw new ArgumentOutOfRangeException();
        }
        // Выполнить какую-то полезную работу
    }
}
```

Исполняющая система может также генерировать исключения как побочный эффект от выполнения кода. Примером сгенерированного системой исключения может служить `NullReferenceException`, которое возникает при попытке обращения к полю или вызова метода на объекте, когда фактически ссылка на объект не существует.

Когда исключение сгенерировано, исполняющая система начинает искать в стеке соответствующий этому исключению блок `catch`. Проходя по стеку, она раскручивает его, очищая по пути каждый фрейм.

Если поиск завершается в последнем фрейме потока, а обработчик исключения не найден, в этой точке исключение считается необработанным. В .NET 2.0 эта проблема решена за счёт требования, чтобы любое необработанное исключение, кроме `AppDomainUnloadException` и `ThreadAbortException`, вызывало прерывание потока. Если поток прерывается, как и должен, это означает генерацию красного флажка в точке исключения, что позволяет немедленно обнаружить проблему и устранить её.

Обзор синтаксиса операторов `try`, `catch` и `finally`

Код внутри блока `try` защищён от исключений так, что если исключение сгенерировано, то исполняющая система ищет подходящий блок `catch`, чтобы его обработать. Независимо от того, существует или нет подходящий блок `catch`, если предусмотрен блок `finally`, он всегда выполняется, независимо от того, как поток управления покидает блок `try`. Рассмотрим пример оператора `try` в C#:

```
using System;
using System.Collections;

public class EntryPoint
{
    static void Main()
    {
```

```
        try
        {
            ArrayList list = new ArrayList();
            list.Add(1);
            Console.WriteLine("Элемент 10 = {0}", list[10]);
        }
        catch (ArgumentOutOfRangeException x)
        {
            Console.WriteLine("=== Обработчик ArgumentOutOfRangeException ===\n\n");
            Console.WriteLine(x);
            Console.WriteLine("=== Обработчик ArgumentOutOfRangeException ===\n\n");
        }
        catch (Exception x)
        {
            Console.WriteLine("=== Обработчик исключения ===");
            Console.WriteLine(x);
            Console.WriteLine("=== Обработчик исключения ===\n\n");
        }
        catch
        {
            Console.WriteLine("=== Необработанное исключение Handler ===");
            Console.WriteLine("Исключение, которое неожидалось...");
            Console.WriteLine("=== Необработанное исключение Handler ===");
        }
        finally
        {
            Console.WriteLine("Очистка...");
        }
    }
}
```

Код внутри блока `try` предназначен для генерации исключения `ArgumentOutOfRangeException`. Как только исключение сгенерировано, исполняющая система начинает поиск подходящей конструкции `catch`, являющейся частью этого оператора `try` и максимально соответствующей типу исключения. Ясно, что лучше всего подходит первая конструкция `catch`. Поэтому исполняющая система немедленно начинает выполнение операторов из первого блока `catch`. Если действительное содержимое исключения не интересует, объявление переменной исключения x в конструкции `catch` можно опустить и ограничится объявлением типа.

Вторая конструкция `catch` будет перехватывать исключения общего типа `Exception`. Если код в блоке `try` сгенерирует исключение, производное от `System.Exception` и отличающееся от `ArgumentOutOfRangeException`, то этот второй блок `catch` обработает его. В C# множественные конструкции `catch`, ассоциированные с одним блоком `try`, должны следовать в таком порядке, чтобы наиболее специфичные исключения обрабатывались первыми. Код, в котором общие конструкции `catch` предшествуют более специфичным, просто не скомпилируется. В этом легко убедиться, поменяв местами первые две конструкции `catch` в предыдущем примере — будет выдано сообщение об ошибке.

В C# каждое исключение, которое можно сгенерировать, должно наследоваться от `System.Exception`. Поскольку объявлена конструкция `catch`, специально предназначенная для исключений типа `System.Exception`, как насчёт третьей и последней конструкции catch? Несмотря на то что сгенерировать исключение типа, не унаследованного от `System.Exception`, в языке C# невозможно, это не является невозможным для среды CLR. (Например, в языке C++ можно сгенерировать исключение любого типа.) Поэтому если написать `ArrayList` на языке, который позволяет это, может получиться, что код сгенерирует исключение не очень полезного типа, такого как `System.Int32`. Звучит странно, но такое возможно. В этом случае можно перехватить такое исключение в C#, применив блок `catch` без явного типа исключения и без переменной. К сожалению, при этом не существует простого способа узнать тип сгенерированного исключения. К тому же оператор `try` может иметь максимум одну общую конструкцию `catch` без аргументов.

В самом конце находится блок `finally`. Независимо от того, как произошел выход из блока `try` — по достижении его конечной точки, через генерацию исключения или оператор `return` — блок `finally` выполняется всегда. Если есть подходящий блок `catch` и блок `finally`, ассоциированный с тем же самым

блоком try, блок catch выполняется перед блоком finally. В этом легко убедиться, взглянув на вывод предыдущего кода примера, который выглядит следующим образом:

```
=== Обработчик ArgumentOutOfRangeException ===


System.ArgumentOutOfRangeException: Индекс за пределами диапазона. Индекс должен
 быть положительным числом, а его размер не должен превышать размер коллекции.
Имя параметра: index
   в System.Collections.ArrayList.get_Item(Int32 index)
   в EntryPoint.Main() в d:\Projects\TestApp\TestApp\Program.cs:строка 12
=== Обработчик ArgumentOutOfRangeException ===


Очистка...
```

Повторная генерация и трансляция исключений

Внутри определённого фрейма стека может понадобиться перехватить все исключения или же определённое их подмножество, выполнить некоторую очистку и затем заново сгенерировать исключение, чтобы позволить ему дальше распространяться по стеку. Для реализации сказанного используется оператор throw без параметров:

```
using System;
using System.Collections;

public class Entrypoint
{
    static void Main()
    {
        try
        {
            try
            {
                ArrayList list = new ArrayList(); list.Add(1);
                Console.WriteLine("Элемент 10 = {0}", list[10]);
            }
            catch (ArgumentOutOfRangeException)
            {
                Console.WriteLine("Выполнить полезную работу и повторить исключение");
                // Заново сгенерировать перехваченное исключение
                throw;
            }
            finally
            {
                Console.WriteLine("Очистка...");
            }
        }
        catch
        {
            Console.WriteLine("Готово");
        }
    }
}
```

Обратите внимание, что любые блоки finally, связанные с фреймом исключения, с которым ассоциирован блок catch, будут выполнены перед выполнением обработчиков любых исключений более высокого уровня.

Иногда необходимо «транслировать» исключение внутри обработчика исключений. В этом случае перехватывается исключение одного типа, но затем генерируется исключение другого типа — возможно, более точного — в блоке `catch` для передачи его на обработку на следующем уровне. Рассмотрим приведённый ниже пример:

```
using System;
using System.Collections;

public class MyException : Exception
{
    public MyException(String reason, Exception inner)
        : base(reason, inner)
    {
    }
}

public class Entrypoint
{
    static void Main()
    {
        try
        {
            try
            {
                ArrayList list = new ArrayList();
                list.Add(1);
                Console.WriteLine("Элемент 10 = {0}", list[10]);
            }
            catch (ArgumentOutOfRangeException x)
            {
                Console.WriteLine("Выполнить полезную работу и повторить исключение");
                throw new MyException("Лучше сгенерировать исключение", x);
            }
            finally
            {
                Console.WriteLine("Очистка...");
            }
        }
        catch (Exception x)
        {
            Console.WriteLine(x);
            Console.WriteLine("Готово");
        }
    }
}
```

Одним особым качеством типа `System.Exception` является способность включать в себя ссылку на вложенное исключение через свойство `Exception.InnerException`. Таким образом, когда генерируется новое исключение, вы можете предохранить цепочку исключений для исключений, обрабатывающих их. Рекомендуется использовать это полезное свойство стандартного типа исключений C# для трансляции собственных исключений. Вывод предыдущего кода выглядит следующим образом:

```
Выполнить полезную работу и повторить исключение
Очистка...
MyException: Лучше сгенерировать исключение ---> System.ArgumentOutOfRangeException:
 Индекс за пределами диапазона. Индекс должен быть положительным числом, а его размер не должен
 превышать размер коллекции.
Имя параметра: index
   в System.Collections.ArrayList.get_Item(Int32 index)
   в Entrypoint.Main() в d:\Projects\TestApp\TestApp\Program.cs:строка 20
   --- Конец трассировки внутреннего стека исключений ---
   в Entrypoint.Main() в d:\Projects\TestApp\TestApp\Program.cs:строка 25
```

Готово

Имейте в виду, что трансляции исключений по возможности следует избегать. Чем больше исключений перехватывается и генерируется заново в стеке, тем больше код, обрабатывающий исключение, будет изолирован от кода, его генерирующего. То есть, становится трудным сопоставить точку перехвата с исходной точкой генерации исключения. Хотя свойство `Exception.InnerException` помогает смягчить проблему, но всё равно трудно найти изначальную причину проблемы, если исключения транслируются по пути.

Исключения, сгенерированные в блоке `finally`

Возможно, но крайне нежелательно, генерировать исключения внутри блока `finally`. Следующий код демонстрирует пример:

```
using System;
using System.Collections;

public class Entrypoint
{
    static void Main()
    {
        try
        {
            try
            {
                ArrayList list = new ArrayList();
                list.Add(1);
                Console.WriteLine("Элемент 10 = {0}", list[10]);
            }
            finally
            {
                Console.WriteLine("Очистка...");
                throw new Exception("Лучше сгенерировать исключение");
            }
        }
        catch (ArgumentOutOfRangeException)
        {
            Console.WriteLine("Аргумент вышел за допустимые пределы!");
        }
        catch
        {
            Console.WriteLine("Готово");
        }
    }
}
```

Первое исключение, `ArgumentOutOfRangeException`, просто теряется, а новое исключение распространяется по стеку. Ясно, что это нежелательно. Никогда не теряйте след исключений, поскольку тогда практически невозможно определить, что именно вызвало исключение в самом начале.

Исключения, сгенерированные в статических конструкторах

Если исключение сгенерировано, а в стеке нет обработчика, поэтому его поиск завершается в статическом конструкторе типа, то исполняющая система обрабатывает этот случай специальным образом. Она транслирует исключение в `System.TypeInitializationException` и генерирует его взамен первоначального.

Перед генерацией нового исключения свойство `InnerException` экземпляра `TypeInitializationException` устанавливается в исходное исключение. Таким образом, любой обработчик исключения инициализации типа может легко обнаружить причину сбоя.

Такая трансляция исключения имеет смысл, потому что конструкторы по своей природе не могут возвращать значение, свидетельствующее об успехе или неудаче. Исключения — единственный доступный механизм, который позволяет сигнализировать о сбое конструктора. Что более важно: поскольку система вызывает статические конструкторы в определённое самой системой время, для них имеет смысл применять

тип TypeInitializationException, который позволит точнее определить, когда что-то идет не так. Например, предположим, что имеется статический конструктор, который может потенциально генерировать исключение ArgumentOutOfRangeException. Теперь представьте разочарование пользователей, если исключение распространится во включающем потоке в некоторый случайный момент времени, поскольку точный момент вызова статического конструктора определяется системой. Может показаться, что ArgumentOutOfRangeException материализуется буквально из ничего. Помещение исключения в оболочку TypeInitializationException немного проясняет ситуацию и предупреждает пользователей, или, надо надеяться, разработчика, о том, что проблема возникла во время инициализации типа.

В приведённом ниже коде демонстрируется пример того, как выглядит TypelnitializationException с вложенным внутри него исключением:

```
using System;
using System.IO;
class EventLogger
{
    static private StreamWriter eventLog;
    static private string strLogName;

    static EventLogger()
    {
        eventLog = File.CreateText("logfile.txt"); // Следующий оператор сгенерирует исключение
        strLogName = (string)strLogName.Clone();
    }

    static public void WriteLog(string someText)
    {
        eventLog.Write(someText);
    }
}

public class EntryPoint
{
    static void Main()
    {
        EventLogger.WriteLog("Зарегистрировать это!");
    }
}
```

В результате запуска этого кода будет получен вывод с указанием того, что внешнее исключение имеет тип TypelnitializationException, а также с указанием внутреннего исключения, с которого все началось — NullReferenceException.

Создание пользовательских классов исключений

У System.Exception имеются три общедоступных конструктора и один защищённый. Первый — это конструктор по умолчанию, который на самом деле мало что делает. Второй — конструктор, принимающий ссылку на строковый объект. Строка представляет собой общее, определяемое программистом сообщение, которое можно рассматривать как более дружественное к пользователю описание исключения. Третий конструктор также принимает строку сообщения, как и второй, но вдобавок принимает ссылку на другой объект Exception. Ссылка на другое исключение позволяет отслеживать исходные исключения, когда внутри блока try одно исключение транслируется в другое. Хорошим примером может служить ситуация, когда исключение не обрабатывается, а просачивается вверх, во фрейм стека статического конструктора. В этом случае исполняющая система генерирует исключение TypeInitializationException, но только после установки внутреннего исключения в исходное исключение, чтобы тот, кто перехватывает TypeInitializationException, по крайней мере, знал, чем вызвано исключение изначально.

Благодаря этим трём общедоступным конструкторам, класс System.Exception очень полезен. Однако простую генерацию объектов типа System.Exception всякий раз, когда в программе что-то идёт не так, следует

считать плохим дизайном. Вместо этого имеет смысл создать новый, более специфичный тип исключения, унаследовав его от `System.Exception`. Таким образом, тип исключения будет более выразительным в описании вызвавшей его проблемы. Ещё лучше то, что производный класс может содержать данные, которые соответствуют причине генерации данного исключения. И помните, что в C# все исключения должны наследоваться от `System.Exception`.