Управление состоянием

# Синтаксис.

# Try блок.

В этом блоке содержится код, который требует некоторых операций по очистке или/и восстановления системы в случае возникновения исключения. То есть главным критерием для кода, который необходимо разместить внутри блока *Try*, является не то, что этот код может или нет бросить исключение, а 2 других важных вопроса – производит ли код такие операции, которым требуется очистка, а также можно ли в данном месте кода провести операцию по восстановлению работоспособности программы после появления исключения.

Часто возникает вопрос, сколько кода нужно размещать внутри блока *Try* – размещать всё в одном блоке или необходимо разбивать на мелкие? Вопрос часто решается индивидуально. Если код может бросить в разных местах одинаковое исключение, причем в разных местах требуется различное поведение по восстановлению состояния системы, то необходимо разбивать код на более мелкие блоки.

# Catch блок.

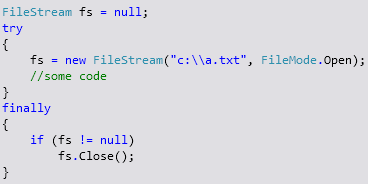
Данный блок содержит код, который необходимо выполнить для обработки исключения. Один блок *try* может содержать несколько блоков c*atch*, но каждый из них должен определять свой собственный тип исключения, который этот блок должен обработать. Если исключение обрабатывается не полностью, а только происходит очистка кода в блоке *try*, то следует в конце блока *catch*снова бросить то же исключение. Причем нужно использовать оператор *throw* без параметров, в этом случае не потеряется стек вызовов.

# Finally блок.

Данный блок содержит код, который гарантировано выполнится. Исключением из этого правила являются следующие причины:

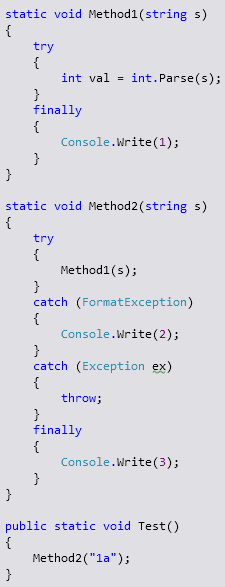
* Нить, в которой вызывался блок, был закрыта методом *TerminateThread*. Если нить была закрыта с помощью метода *Thread.Abort*, то будет брошено исключение *ThreadAbortException*, которое не прервет выполнение кода внутри блока *finally*.
* Вызов метода *Environment.FailFast*. Метод немедленно прерывает выполнение процесса, записав сообщение об ошибке в лог системы.
* Вызов *TerminateProcess*.

Как правило, блок *finally* содержит код по очистке ресурсов, которые задействуются внутри блока *try*.



Типичное использование. Перед блоком *try* переменной, управляющая некоторым ресурсом, присваивается *null*, в блоке *try* переменная управляет ресурсом. В блоке *finally* проверяется, что переменная не нул, то есть ей передали управление ресурсом, после чего ресурс освобождается.

# Порядок выполнения блоков.



Рассмотрим пример. Метод тест выполняет метод *Method2*, который в свою очередь выполняет метод *Method1*, внутри которого бросается исключение типа *FormatException*. После бросания исключения выполнится блок *finally* в методе *Method1*, в котором бросается исключение. Здесь нет блока *catch*, поэтому исключение проходит в метод *Method2*. В нем исключение ловится первым блоком *catch*, поскольку совпадают типы исключений. После отработки исключения отрабатывает блок *finally*.

# System.Exception.

CLR разрешает бросать в качестве исключения объекты любого типа, например, *Int32*. Однако язык C# ограничивает эту возможность, разрешая бросать объекты типов, пронаследованных от *System.Exception*. Данный тип содержит несколько очень полезных для отладки свойств, таких как *StackTrace*, *Source* и прочее. Стоит иметь в виду, что в релизной версии JIT-компилятор может оптимизировать код таким образом, что некоторые методы будут откомпилированы inline, и в *StackTrace* такие методы могут отсутствовать.

# Определяем понятие исключения.

Как правило, имя метода – это некий глагол, то есть то, что этот метод должен сделать. Если метод не может выполнить ту задачу, которая ей определяется этим названием, то метод должен кинуть исключение. Существует другой способ сигнализировать вызвавшему, что метод не смог выполнить свою задачу – возвращение кода ошибки. Данный способ применялся первым, и его главным недостатком было то, что разработчики редко проверяли код возврата функций, кроме того, проверки всех функций на код возврата сильно ухудшал читаемость кода.

В .NET сделана ставка на обработку ошибок с помощью исключений. Поэтому следует избегать использование кода ошибок, поскольку сочетание обоих способов еще более ухудшает и читаемость кода, и управляемость при появлении ошибок.

Следует иметь в виду, что появление исключения – сигнал о некорректной работе программы, и в нормальной ситуации исключения не должны появляться совсем. Если при использовании метода появление исключений подразумевает нормальную работу, стоит использовать другой вариант метода, который пытается сделать что-то, а не делает. В таком случае название метода должно быть чем-то вроде *TryDo* и возвращать булевское значение. Так, например, существуют методы *int.Parse* и *int.TryParse*. Однако использовать вариант *«Try»* следует только тогда, когда заранее известно, что метод может не выполнить свою задачу, и это нормальная ситуация. Например, есть массив строк, среди которых встречаются строки с целыми числами внутри, и мы хотим получить все такие значения. Здесь исключения при попытке парсинга не совсем уместны.

# Модели работы с исключениями.

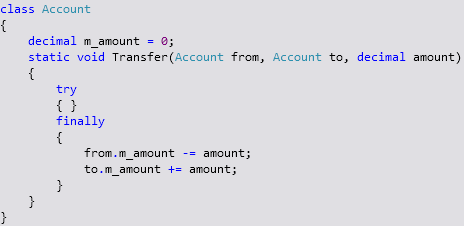
Проектируя продукт или библиотеку, очень важно решить, как будут обрабатываться исключения. Особенно это касается разработки библиотеки классов. Здесь необходимо строго следить за всеми публичными классами и методами в них и давать единообразный по модели и максимально информативный ответ на возникающие ошибки. Если разрабатывается свое собственное приложение, то здесь важно сформулировать единую методологию генерации и обработки исключений.

Существует несколько общих моделей и рекомендаций по методологии исключений.

# Чаще используйте блок *finally*.

Блок *finally* очень полезен. Этот код гарантируется к исполнению, поэтому именно в этом блоке рекомендуется производить все закрытия ресурсов.

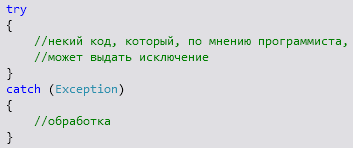
Другой полезной особенностью данного блока является то, что выполнение кода в нем не прерывается при вызове *Thread.Abort*, так что в этот блок помещать важные изменения значений некоторых переменных, например, следующие:



Здесь метод *Account.Transfer* переводит условную сумму денег с одного счета на другой. Если между вычитанием из первого счета и добавлением во второй будет брошено исключение с помощью *Thread.Abort*(и код не был бы внутри блока *finally*), то произошла бы серьезная ошибка в состоянии системы. Из одного счета уже произошло вычитание средств, а в другой – они не были зачислены. Дальнейшее обращение к этим объектам привело бы к критичной ошибке. То, что этот код находится внутри данного блока позволяет исключить такую ошибку. Исключение будет брошено уже за пределами блока *finally*.

# Не ловите все исключения.

Очень часто работа с исключениями идет по следующей схеме:



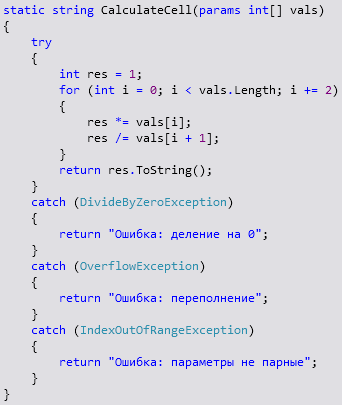
То есть программист пишет некоторый код, в котором, как считает программист, при вполне обычных, то есть не экзотических, случаях может бросаться исключение. Такую часть кода программист оборачивает в блок *try*, далее он создает только один блок с *catch*, который ловит все исключения, даже если программист ожидает вполне конкретный тип исключения (например, при работе с базой данных, он может ожидать, что при выключенном sql-сервере будет брошено исключение типа *SqlException*). Все остальные исключения он ловит за между делом, на всякий случай. Кроме того, очень часто, не особенно понимая, что же делать с этой ошибкой, она просто логгируется, а функция продолжает свою работу, как ни в чем не бывало. Или прерывается, но только сама функция.

Вторая ошибка здесь в том, что если было поймано не то исключение, которое ожидает программист, такое как *System.OutOfMemoryException*, то обработка исключения по тому же алгоритму, что и *SqlException*, может сильно испортить состояние системы.

Таким образом, важное правило – «не ловите все». Ловите исключение там, где вы его реально можете обработать, восстановив систему для дальнейшей работы. Разделяйте исключения по типу, ошибку разных типов можно обработать на разных уровнях приложения так, чтобы не прерывать процедуру там, где этого можно избежать.

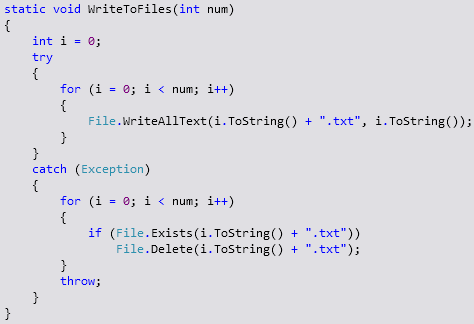
В тех же случаях, когда нужно поймать именно все исключения, будьте уверены, что оставшиеся в употреблении переменные будут находиться в корректном состоянии при всех возможных ошибках. Использование переменных, находящихся в некорректном состоянии – ошибка более серьезная, чем падение системы.

# Восстановление согласно исключению.



В данном примере функция подготавливает строчку для отображения в ячейке таблицы. Для этого она производит некоторые вычисления. В случае ошибок в вычислении функция вместо результата вычислений возвращает строчку для пользователя с кратким описанием ошибки.

# Откат частично выполненных операций.



В данном примере пишется в несколько фалов некоторые данные. Если в процессе работы функции произошла некая ошибка, необходимо почистить состояние системы, в данном случае удалить записанные файлы. Поскольку это не полное восстановление системы после ошибки, то в конце мы бросаем исключение выше по стеку.

# Сокрытие деталей реализации для соблюдения модели исключений.



В данном примере ищется телефон по имени. Здесь в процессе выполнения кода могут быть брошены несколько разных исключений, однако они сами по себе мало помогут пользователям данного метода, поэтому вместо них бросается более понятное для пользователя, но добавляя оригинальное исключение в качестве внутреннего.