## 13. Исключения и их объектная обработка

**Исключения** (**exception**) – реакция вычислительной системы на некоторую **особую ситуацию**, возникшую при работе приложения. Такие особые ситуации связаны с возникновением в исполняемом коде программы какой-либо ошибки (**run–time error**, ошибки времени выполнения), например - обращение к несуществующему файлу, неправильная работа с адресным указателем, несоответствие типов данных и т.д. В этом случае программа не может дальше выполняться и аварийно завершается с выводом некоторого кода ошибки.

Любая современная программа должна уметь **перехватывать** такие ситуации и, по возможности, обрабатывать их тем или иным способом. Для этого обычно в программу вставляется большое число проверок разных условий, что делает логику программы достаточно запутанной. Механизм исключений позволяет упростить обработку особых ситуаций, выделив в программе специальные блоки - обработчики исключений.

Все основные современные операционные системы и языки программирования поддерживают **централизованную** обработку исключений, которая строится на **объектных** принципах. Объектная обработка особых ситуаций основана на двух основных моментах:

- все типовые особые ситуации **классифицированы** и формализованы в виде специальных **стандартных классов** базовых библиотек
- при возникновении особой ситуации **автоматически** создается **объект** соответствующего класса, который несет информацию о возникшей проблеме

Стандартные классы исключений построены по иерархическому принципу в полном соответствии с тезисом «от общего к частному». Корнем иерархии является абстрактный класс **Exception**, от которого порождаются дочерние подклассы для различных типов ошибочных ситуаций. Например, платформа .NET Framework поддерживает около 100 стандартных классов, некоторые из которых представлены на следующей схеме:

**Object** // корневой класс всей иерархии классов

**Exception** // корень подиерархии классов для исключений

**SystemException** // имеет около 70 непосредственных потомков

ArithmeticException // ошибки арифметических операций

DivideByZeroException // попытка деления на ноль

OverflowException // арифметическое переполнение

ArrayTypeMismatchException // несоответствие типов для массива

IndexOutOfRangeException // выход индекса за границы массива

InvalidCastException // ошибки при преобразовании типов

IOException // ошибки при выполнении операций ввода/вывода

EndOfStreamException //попытка чтения за концом входн. потока

FileNotFoundException // обращение к несуществующему файлу

NullReferenceException // использование пустого указателя

OutOfMemoryException // программе не хватает памяти

StackOverflowException // переполнение стека вызовов подпрограмм

Аналогичный вид имеют подиерархии классов исключений для платформы Java и пакета Delphi. Для сравнения можно привести несколько стандартных классов исключений, входящих в базовую библиотеку классов Delphi:

- **EInOutError** // ошибки при выполнении операций ввода/вывода
- **EIntError** // ошибки целочисленной арифметики
- **EMathError** // ошибки при обработке вещественных чисел
- **EConvertError** // ошибки преобразования типов
- **EAccessViolation** // ошибки неправильного использования памяти

Знание иерархии классов исключений и умение использования этого механизма является очень важным аспектом разработки современного надежного программного обеспечения, поскольку оно позволяет реализовать один из основных принципов квалифицированной разработки: «Хорошая программа никогда не должна завершаться аварийно».

Перейдем к вопросу **использования** механизма исключений в объектных программах. Для этого, прежде всего, в исходном тексте необходимо выделить те операторы, которые **потенциально** могут быть источниками ошибок времени выполнения. Далеко не каждый оператор программы является потенциально опасным, поэтому очень важно знать источники возможных неприятностей. База знаний по этим источникам – упомянутые выше стандартные классы исключений.

Потенциально опасные операторы могут идти последовательно друг за другом, либо могут быть разбросаны по всему исходному коду. Главное – **найти** такие операторы и **изолировать** их в тексте программы с помощью специальных синтаксических конструкций. Начало потенциально опасного фрагмента программы во **всех** объектных языках принято обозначать директивой **try**. За этой директивой должен следовать потенциально опасный фрагмент кода, за которым размещается специальный блок обработчиков исключений. Начало этого блока обозначается директивой **catch** в языках Java и C# и директивой **except** в языке Delphi Pascal. Общий вид защищенного от ошибок времени выполнения кода выглядит следующим образом:

| | |
|---|---|
| **try**<br>{ опасные операторы }<br>**catch**<br>{ обработка исключения } | **try**<br>опасные операторы<br>**except**<br>обработчики исключений<br>**end**; |

Такой защищенный код работает следующим образом:

- если при его выполнении исключения не возникают, то блок **catch/except** вообще не выполняется, и управление передается операторам, следующим за этим блоком

- если же в защищенном коде возникает исключение, то оставшиеся операторы защищенного кода не выполняются, а управление сразу передается на блок обработки.

Простейшая обработка особой ситуации может лишь включать вывод информационного сообщения, но даже это позволяет **не прерывать аварийно** выполнение программы, а **информировать** пользователя о возникшей проблеме и дать ему возможность как-то **прореагировать** на эту ситуацию.

**Пример**: организация защиты от особой ситуации «Попытка деления на ноль». Фрагмент программы с одним защищенным оператором:

| | |
|---|---|
| . . . . . // предыдущие операторы<br>**try** // начало<br>{ z = x / y; } // опасный оператор<br>**catch**<br>{ Console.WriteLine(“Что-то не так ”);<br>} // конец<br>. . . . . // следующие операторы | . . . . . // предыдущие операторы<br>**try** // начало<br>z := x / y; // опасный оператор<br>**except**<br>ShowMessage(‘Ошибка, однако’)<br>**end**; // конец<br>. . . . . // следующие операторы |

Еще раз подчеркнем, что если при делении особой ситуации не возникнет, то обработчики не активизируются, никакие сообщения не будут выведены и выполнение программы продолжится операторами за блоком обработки.

На практике рекомендуется использовать везде, где это возможно более **конкретный** разбор возникшей проблемы. Для этого при оформлении блока обработки можно указывать **тип** обрабатываемого исключения, используя для этого **имя** соответствующего **класса**. Это имя задается в круглых скобках после директивы **catch** или как часть Delphi-конструкции вида

**on** ИмяКласса **do** оператор;

В качестве примера рассмотрим фрагмент программы, в цикле выполняющей поэлементное деление двух массивов. Здесь опасной опять же является операция деления. Защиту такого кода можно организовать по-разному. Можно защитить **весь** цикл деления, а можно – **отдельно** каждую операцию деления внутри цикла. Результаты работы таких программ будут разными:

- в первом случае выполнение цикла будет прервано при **первой** попытке деления на ноль и остальные элементы массивов **не будут** обработаны
- во втором случае цикл **всегда** будет выполнен **полностью**, просто для нулевых элементов массива-делителя будут зафиксированы особые ситуации, они будут обработаны блоком обработки и это даст возможность продолжить работу цикла.

Второй вариант интересен тем, что он демонстрирует один из основных принципов механизма исключений, а именно: если исключение обработано программой, соответствующий объект уничтожается, и программа может продолжить свое выполнение. Далее приводятся фрагменты соответствующих программ (для краткости массивы имеют имена A и B).

| | |
|---|---|
| // Вариант 1.<br>**try { //** защищенный фрагмент-цикл<br>**for** (**int** i=0; i<10; i++)<br>{ A[i] = A[i] / B[i]; }<br>}<br>**catch** (DivideByZeroException)<br>{ Console.WriteLine(“Делитель = 0”);<br>} // конец<br>. . . . . // следующие операторы | // Вариант 1.<br>**try //** защищенный фрагмент-цикл<br>**for** i:= 1 **to** 10 **do**<br>A[i] := A[i] / B[i];<br><br>**except on** EDivByZero **do**<br>ShowMessage(‘Делитель = 0’)<br>**end**; // конец<br>. . . . . // следующие операторы |
| // Вариант 2.<br>**for** (**int** i=0; i<10; i++) { | // Вариант 2.<br>**for** i:= 1 **to** 10 **do** |

| | |
|---|---|
| **try** // защищена операция<br>{ A[i] = A[i] / B[i]; }<br>**catch** (DivideByZeroException)<br>{ Console.WriteLine(“Делитель = 0”);<br>}<br>. . . . . // следующие операторы | **try** // защищена операция<br>A[i] := A[i] / B[i];<br>**except on** EDivByZero **do**<br>ShowMessage(‘Делитель = 0’)<br>**end**;<br>. . . . . // следующие операторы |

Довольно часто в программах встречаются расположенные рядом опасные операторы **разных** типов. Защищать блоком **try** каждый из них в отдельности достаточно громоздко. В этом случае можно оформить **один try**-блок сразу на **группу** операторов, а в блоке обработки предусмотреть возможность реагирования на исключения **различных** типов. Блок обработки в этом случае оформляется следующим образом:

| | |
|---|---|
| **catch** (ИмяКласса1)<br>{ обработка исключения типа 1}<br>**catch** (ИмяКласса2)<br>{ обработка исключения типа 2}<br>**catch** (ИмяКласса3)<br>{ обработка исключения типа 3 }<br>. . . . . | **except**<br>**on** ИмяКласса1 **do** обработка1;<br>**on** ИмяКласса2 **do** обработка2;<br>**on** ИмяКласса3 **do** обработка3;<br>. . . . . .<br>**end**; |

В этом случае важным является **порядок** следования обработчиков: сначала указываются обработчики наиболее конкретных типов, а потом - более общих типов. Это связано с тем, что при возникновении исключения поиск подходящего обработчика выполняется по **порядку** их следования в блоке и в соответствии с **иерархической** принадлежностью. Как только будет найден первый **подходящий** обработчик, он выполняется а остальные просто пропускаются.

Вот пример **правильной** организации списка обработчиков:

| | |
|---|---|
| **catch** (DivideByZeroException) | **except** |

| | |
|---|---|
| {обработка ошибки деления на 0}<br>**catch** (ArithmeticException )<br>{обработка других арифм. ошибок}<br>**catch** (Exception )<br>{ обработка всех остальных ошибок любого типа (корень!)} | **on** EDivByZero **do** . . . . .<br>// обработка ошибки деления на 0<br>**on** EIntError **do** . . . . .<br>// обработка других арифм. ошибок<br>**on** Exception **do** . . . . .<br>// обработка всех остальных ошибок<br>**end**; |

Если в защищенном коде возникает ошибка, обработчик которой **не предусмотрен** в блоке **catch/except,** то автоматически вызывается **стандартный** системный обработчик, который выводит общее сообщение и **аварийно** завершает работу.

Наиболее полный анализ возникшей особой ситуации предоставляет автоматически создаваемый объект-экземпляр некоторого класса исключения. Для получения доступа к этому объекту надо просто вместе с именем класса объявить еще и имя объектной переменной:

| |
|---|
| **catch** (ИмяКласса ИмяОбъекта)<br>{ обработка с использованием переменной ИмяОбъекта } |
| **except on** (ИмяОбъекта : ИмяКласса) **do**<br>// обработка с использованием переменной ИмяОбъекта |

Иногда возникает необходимость программно сгенерировать некоторое исключение. Для этого можно использовать специальные операторы:

- в языках Java, C# и C++ оператор **throw**
- в Delphi - оператор **raise**

Синтаксически использование этих операторов похоже на создание объектов, что, впрочем, и происходит на самом деле: надо указать имя класса исключений и один из его конструкторов:

- **throw new** ИмяКласса (параметры конструктора)
- **raise** ИмяКласса.Create (параметры)

Хотя стандартных типов исключений достаточно для подавляющего числа практических ситуаций, при необходимости можно создать и **собственные** нестандартные исключения. Для этого достаточно создать новый класс как **производный** от базового класса **Exception**, используя стандартный синтаксис объявления дочернего класса.

В завершение данного раздела отметим еще одну полезную возможность, реализованную во **всех** языках: кроме стандартного блока типа **try/catch** или **try/except** можно использовать конструкцию типа **try/finally.** Она позволяет оформить так называемый **блок завершения**: набор операторов, которые будут выполняться **всегда**, независимо от возникновения исключения. Блок завершения целесообразно использовать для освобождения занятых ресурсов, например – освобождение динамически выделенной памяти, закрытие файлов и т.д. Очень часто блоки обработки исключений и завершения объединяются вместе в одну конструкцию, причем сначала оформляется блок обработки, а затем – блок завершения.

| | |
|---|---|
| **try { ** | **try** |
| . . . . . } | . . . . . |
| **catch** ( . . . . . ) { . . . . . } | **except** |
| **catch** ( . . . . . ) { . . . . . } | **on** . . . . **do** . . . . . |
| **catch** ( . . . . . ) { . . . . . } | **on** . . . . **do** . . . . . |
| **finally** { код завершения } | **finally** // код завершения |
| | **end**; |