## Тема 4. Делегаты, события и обработка исключений

**4.1. Делегаты – специальные классы**

**4.1.1. Описание и использование делегатов**

**4.1.2 Групповые (многоадресные) делегаты**

**4.2. События – члены класса**

**4.3. Обработка исключений в среде .NET Framework**

**4.4. Задачи для самостоятельного решения**

**4.5. Лабораторная работа по теме «*Делегаты, события и обработка***

***исключений*»**

**4.6. Контрольные вопросы**

### 4.1. Делегаты – специальные классы

#### 4.1.1. Описание и использование делегатов

***Делегат*** – это вид класса (тип), предназначенный для хранения ссылок на методы. Делегаты применяются, в основном, для следующих целей:

* получения возможности определять вызываемый метод не при компиляции, а динамически, во время выполнения программы;
* создания универсальных методов, в которые можно передавать как параметры другие методы;
* поддержки механизма обратных вызовов, который используется при обработке событий.

Как и всякий класс, делегат представляет собой тип данных и является наследником системного класса **System.Delegate** и наследует от него некоторые полезные методы и свойства. В предыдущих темах уже говорилось о том, что класс можно рассматривать как тип данных, определяемый пользователем. Этот тип определяется в модуле класса, в котором между ключевыми словами **Class...End Class** (т.е. в теле класса), описываются члены класса. После такого определения можно с помощью конструктора создавать объекты (экземпляры) этого типа (класса). Сами объекты (экземпляры) класса создаются вне тела класса.

Для делегатов часто и тип, и экземпляр этого типа называют одним термином «делегат», поэтому нужно учитывать контекст, где этот термин используется. Делегат как тип никогда не имеет никаких членов, да и тела, ограниченного словами **Class...End Class** у него нет, поэтому экземпляр делегата, в отличие от экземпляров, рассмотренных нами ранее классов, не наделяется каким-то участком памяти, а, как уже говорилось, содержит ссылку на метод. В **Теме 2** мы напоминали, что ссылка на объект, по существу, представляет собой адрес этого объекта; таким образом, ссылка на метод – это адрес, по которому программа обращается при вызове этого метода. Этот адрес можно присвоить делегату (экземпляру делегата), и затем использовать делегат для вызова этого метода. Во время выполнения программы один и тот же экземпляр делегата можно использовать для вызова различных методов.

Для объявления делегата как типа используется ключевое слово **Delegate**. Определение (объявление) делегата задает сигнатуру методов, которые могут быть вызваны с его помощью, например:

|  |
| --- |
| *'Определение делегата как типа*  **Public Delegate Function func(ByVal a As Integer) As String**  **Public Delegate Sub proc(ByVal t As String, ByVal r As String)** |

Здесь описаны два разных типа делегата: первый может хранить ссылки на методы, которые принимают один параметр целого типа и возвращают строковое значение, второй – на методы, принимающие в качестве параметров две строки и ничего не возвращающие. После определения сигнатуры делегата можно создавать код методов, которые делегат будет вызывать. Эти методы могут вызываться только в тех случаях, когда их сигнатура соответствует сигна­туре делегата (то есть методы и делегат должны иметь параметры и возвращаемые значения оди­наковых типов). Если это условие не будет выполнено, при компилировании программного кода возникнет ошибка компиляции.

Для того, чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. Таким образом, выделяется три этапа применения делегата: объявление (определение) делегата как типа (причем определение делегата не создает его экземпляр, а только вводит тип, на основе которого позже могут быть созданы экземпляры), создание экземпляра делегата и обращение к экземпляру делегата (вызов делегата).

Так как делегат – это ссылочный тип, то после определения делегата как типа, можно создавать переменные с типом делегата, т.е. – ссылки на него, например:

|  |
| --- |
| *'Определение ссылок на делегат*  **Dim** **D1** **As** **func** *'ссылка с типом делегата**func*  **Dim** **D2** **As** **proc** *'ссылка с типом делегата**proc* |

После такого определения ссылки **D1** и **D2** имеют неопределенные значения (**Nothing**). С каждой из этих ссылок можно связать экземпляр соответствующего типа делегатов. Экземпляр создается конструктором делегата, который автоматически встраивается компилятором в объявление делегата как типа:

|  |
| --- |
| *'Создание экземпляра делегата*  **D1 =** **New func** **(AddressOf** **<***параметр***>)**  **D2** **=** **New proc(AddressOf** **<***параметр***>)** |

В качестве параметра можно использовать:

* статический метод класса (имя метода, уточнённое именем класса);
* метод объекта (имя метода, уточнённое именем объекта);
* ссылку на уже существующий в программе экземпляр делегата.

В VB допускается (и применяется гораздо чаще) более короткий способ создания экземпляра делегата без использования **New**:

|  |
| --- |
| *'Создание экземпляра делегата*  **D1 =** **AddressOf** **<***параметр***>** |

Компилятор VB.NET понимает, что делегат создается для объекта или класса, который задан в параметре.

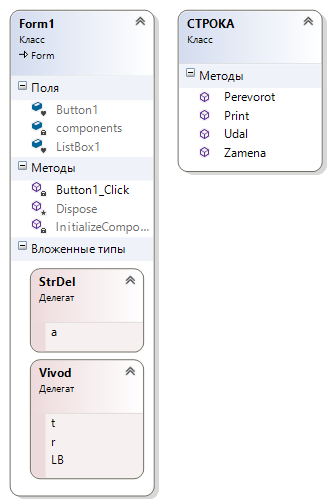
**Пример 4.1-1. Создать проект с классом** **СТРОКА**, **в котором определить три метода для преобразования строки, причем для иллюстрации работы делегата объявить два метода как статические, и один – экземплярный.** Кроме этого, определить в классе статический метод для вывода строк в элемент управления **ListBox**. В модуле формы определить два делегата-типа и в программном коде событийной процедуры привести различные способы создания и использования экземпляров делегатов и их ссылок.

Программный код проекта с классом **СТРОКА** и модулем формы **Form1** приведён на рис. 4.1-1.

|  |
| --- |
| *'Модуль формы*  **Public Class Form1**  *'Объявление двух делегатов - типов*  **Public Delegate Function StrDel(ByVal a As String) As String**  **Public Delegate Sub Vivod(ByVal t As String, ByVal r As String,\_**  **ByRef LB As ListBox)**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim a As String = "МАМА МЫЛА РАМУ"** *'тестируемая строка*  **ListBox1.Items.Add("Исходная строка ")**  **ListBox1.Items.Add(a)**  **Dim rez As String** *'строка - результат*  **Dim Deleg1 As StrDel** *'переменная ссылка на делегат*  **Dim Deleg2 As Vivod** *'ссылка на делегат*  *'экземпляр делегата Deleg1 создается и «привязывается» к*  *'статическому методу Zamena() класса СТРОКА длинным способом*  **Deleg1 = New StrDel(AddressOf СТРОКА.Zamena)**  *'экземпляр делегата Deleg2 создается короче: оператором*  *'AddressOf для адреса статического метода -*  *'процедуры Print() класса СТРОКА:*  **Deleg2 = AddressOf СТРОКА.Print**  **rez = Deleg1(a)** *'вызов метода-функции Zamena() через делегат1*  *'вызов метода-процедуры Print() через делегат2:*  **Deleg2("Замена пробелов звездочками ", rez, ListBox1)**  **Dim OB As New СТРОКА** *'создание объекта класса СТРОКА*  *'экземпляр делегата "привязывается" к экземплярному*  *'методу Udal() объекта OB класса СТРОКА:*  **Deleg1 = AddressOf OB.Udal**  **rez = Deleg1(a)** *'вызов метода через делегат1*  *'вызов метода-процедуры Print() через делегат2:*  **Deleg2("Удаление пробелов ", rez, ListBox1)**  *'экземпляр делегата Deleg1 создается оператором*  *'AddressOf для адреса статического метода -*  *'функции Perevorot() класса СТРОКА:*  **Deleg1 = AddressOf СТРОКА.Perevorot**  **rez = Deleg1(a)** *'вызов метода через делегат1*  *'вызов метода-процедуры Print() через делегат2:*  **Deleg2("Строка наоборот ", rez, ListBox1)**  *' Явное обращение к статическому методу Perevorot()*  *'снова переворачивает полученную строку-результат*  **rez = СТРОКА.Perevorot(rez)**  *' Явное обращение к статическому методу Print()*  **СТРОКА.Print("Опять строка наоборот ", rez, ListBox1)**  **End Sub**  **End Class**  *'Класс с разными методами изменения строки и одинаковой сигнатурой*  **Public Class СТРОКА**  *'статический метод замены пробелов в строке звездочками*  **Public Shared Function Zamena(ByVal a As String) As String**  **Dim t As String**  **t = a.Replace(" ", "\*")**  **Return t**  **End Function**  *'экземплярный метод удаления пробелов из строки*  **Public Function Udal(ByVal a As String) As String**  **Dim t As String = ""**  **For i As Integer = 0 To a.Length - 1**  **If a.Substring(i, 1) <> " " Then**  **t = t + a.Substring(i, 1)**  **End If**  **Next**  **Return t**  **End Function**  *'статический метод переворачивает символы строки наоборот*  **Public Shared Function Perevorot(ByVal a As String) As String**  **Dim t As String = ""**  **For i As Integer = a.Length - 1 To 0 Step -1**  **t = t + a.Substring(i, 1)**  **Next**  **Return t**  **End Function**  *'статический метод вывода строк в ListBox*  **Public Shared Sub Print(ByVal t As String, \_**  **ByVal rez As String,\_**  **ByRef LB As ListBox)**  **LB.Items.Add(t)**  **LB.Items.Add("Результат: " & rez)**  **End Sub**  **End Class** |

*Рис. 4.1-1. Программный код* ***Примера 4.1-1***

*с различными способами создания и использования делегатов*



*Рис. 4.1-2. Схема классов* ***Примера 4.1-1***

*с различными способами создания и использования делегатов*

В модуле формы определены два делегата-типа: функция **StrDel()** с формальным параметром типа **String** и возвращающая результат также типа **String** и процедура **Vivod()**, принимающая два параметра типа **String** и один параметр типа **ListBox**. В классе **СТРОКА** объявлены: статический метод **Zamena()** длязамены пробелов в строке звездочками, экземплярный метод **Udal()** для удаления пробелов из строки, статический метод **Perevorot()**, которыйпереворачивает символы строки наоборот (все три метода имеют ту же сигнатуру, что и делегат – тип **StrDel** и поэтому могут быть представлены экземпляром этого делегата). Также в классе есть статический метод **Print()** для вывода заголовка и строки-результата в элемент управления **ListBox** на форме, который может быть представлен экземпляром делегата **Vivod()**, так как сигнатура этого метода совпадает с сигнатурой второго делегата – типа **Vivod**. В событийной процедуре определены переменные ссылки **Deleg1** и **Deleg2**, которые затем «настраиваются» на экземпляры делегатов **StrDel** и **Vivod**. При создании экземпляров делегатов используются как *параметр*уточнённые имена статических методов **СТРОКА.Zamena** и **СТРОКА.Print**. Теперь ссылка **Deleg1** представляет метод **СТРОКА.Zamena()**, а ссылка **Deleg2** – метод **СТРОКА.Print()**. Поэтому выполнение оператора, записанного в следующей строке программы:

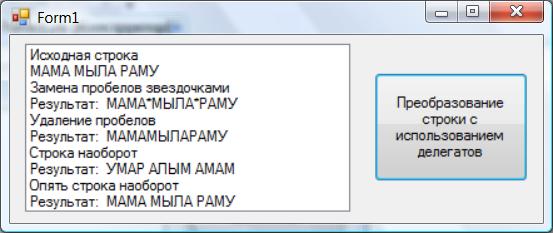
|  |
| --- |
| **rez = Deleg1(a)** |

приводит к вызову метода **Zamena()** класса **СТРОКА** (пробелы заменяются звездочками). Затем делегату **Deleg1** присваивается ссылка на экземплярный метод **Udal()** (после создания объекта **OB** класса **СТРОКА**), и поэтому теперь выполнение точно такого же оператора:

|  |
| --- |
| **rez = Deleg1(a)** |

вызывает метод **Udal()** удаления пробелов из строки. Наконец, переменной **Deleg1** присваивается ссылка на метод **Perevorot()**, и после этого вызов **Deleg1** приводит к вызову этого метода.

Вызов **Deleg2** с тремя фактическими параметрами (первый – заголовок, второй – выводимая строка-результат и третий – **ListBox1**) всегда приводит к вызову статического метода **Print()**.



*Рис. 4.1-3. Результаты работы программы с различными* *способами*   
*создания и использования делегатов*

В двух последних строках программы для иллюстрации выполнено явное (без делегатов) обращение к статическим методам **Perevorot**() и **Print()** класса **СТРОКА**. Результаты работы программы приведены на   
рис. 4.1-3.

#### 4.1.2 Групповые (многоадресные) делегаты

Очень важной особенностью является то, что экземпляр делегата может содержать ссылки сразу на несколько методов, соответствующих типу делегата – это так называемый групповой (многоадресный) делегат. В этом случае при обращении к такому делегату, который хранит ссылки на несколько методов, эти методы вызываются последовательно в том порядке, в котором были добавлены в делегат. Добавление метода (ссылки на метод через делегата) в список выполняется с помощью метода **System.Delegate.Combine (Delegate a, Delegate b, …)**. Удаление ссылки на метод из группового делегата производит метод **System.Delegate.Remove (Delegate** **source, Delegate value)**, в котором первый параметр является групповым экземпляром делегата, а второй – экземпляр делегата, значение которого (ссылку на метод) надо удалить из группового делегата. При добавлении ссылок на методы в групповой делегат происходит **неявное** преобразование типа метода **Combine()** к типу группового делегата, поэтому если в приложении запрещены неявные преобразования (инструкцией **Option Strict On**), то для явного преобразования необходимо воспользоваться функцией **CType()**. То же самое касается и удаления ссылок на методы из группового делегата методом **Remove()**.

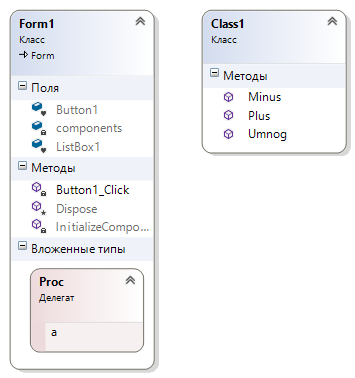
**Пример 4.1-2. Создать класс** **Class1**, **в котором определить три статических метода изменения значения целого числа** (**Plus()** - увеличивает число на 2, **Minus()** уменьшает на 10, **Umnog()** умножает на 3). В модуле формы определить делегат для этих методов и в программном коде событийной процедуры создать экземпляры делегатов и групповой (многоадресный) экземпляр делегата.

Программный код класса и модуля формы приведён на рис. 4.1-4, Схема классов на рис. 4.1-5, а результат работы программы – на рис. 4.1-6.

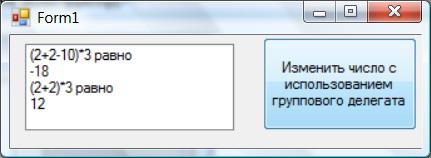
|  |
| --- |
| *'Класс с разными методами изменения целого числа*  *'Три метода имеют одинаковую сигнатуру*  **Public Class Class1**  **Public Shared Sub Plus(ByRef a As Integer)**  **a = a + 2**  **End Sub**  **Public Shared Sub Minus(ByRef a As Integer)**  **a = a - 10**  **End Sub**  **Public Shared Sub Umnog(ByRef a As Integer)**  **a = a \* 3**  **End Sub**  **End Class**  *'Модуль формы*  **Public Class Form1**  *'Объявление делегата-типа*  **Public Delegate Sub Proc(ByRef a As Integer)**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim a As Integer = 2** *'исходное число*  *'Объявление переменных ссылок на делегат:*  **Dim Del, Deleg1, Deleg2, Deleg3 As Proc**  *'создаются экземпляры делегатов и "привязываются" к методам*  **Deleg1 = AddressOf Class1.Plus**  **Deleg2 = AddressOf Class1.Minus**  **Deleg3 = AddressOf Class1.Umnog**  *'создается экземпляр группового делегата*  *'неявное преобразование*  **Del = System.Delegate.Combine(Deleg1, Deleg2, Deleg3)**  **Del(a)** *'вызов группового делегата*  **ListBox1.Items.Add("(2+2-10)\*3 равно")**  *'вывод результата работы группового делегата:*  **ListBox1.Items.Add(a)**  *'Удаление делегата(ссылки на метод)Deleg2*  *' из группового делегата Del:(неявное преобразование)*  **Del = System.Delegate.Remove(Del, Deleg2)**  *'Восстановление исходного значения переменной*  **a = 2**  **Del(a)** *'вызов группового делегата*  **ListBox1.Items.Add("(2+2)\*3 равно")**  *'результат работы измененного группового делегата:*  **ListBox1.Items.Add(a)**  **End Sub**  **End Class** |

*Рис. 4.1-4. Программный код* **Примера 4.1-2**

*с использованием многоадресного делегата*



*Рис. 4.1-5. Схема классов* **Примера 4.1-2**



*Рис. 4.1-6. Результаты работы многоадресного делегата*

При запрещении неявных преобразований строки программы:

|  |
| --- |
| **Del = System.Delegate.Combine(Deleg1, Deleg2, Deleg3)**  **Del = System.Delegate.Remove(Del, Deleg2)** |

заменяются на следующие строки:

|  |
| --- |
| *'Явное преобраз. методов**Combine() и Remove() к типу делегата Proc*  **Del = CType(System.Delegate.Combine(Deleg1, Deleg2, Deleg3), Proc)**  **Del =** **CType(System.Delegate.Remove(Del, Deleg2), Proc)** |

Итак, обращение к экземпляру группового делегата приводит к последовательному выполнению цепочки методов. Результат выполнения цепочки – то значение, которое возвращает последний делегат цепочки.

При вызове цепочки методов через групповой делегат необходимо учесть следующие важные моменты:

* сигнатура методов должна полностью соответствовать делегату;
* методы могут быть статическими и экземплярными;
* каждому методу в списке передается один и тот же набор параметров;
* если параметр передается по ссылке (как в Примере 4.1-2), то изменения параметра в одном методе отразятся на его значении при вызове следующего метода;
* если методом является функция, то результатом выполнения группового делегата является значение, которое возвращает последний метод-функция и поэтому рекомендуется формировать многоадресные делегаты только из методов – процедур.

Использование делегата и цепочки вызовов представляет собой мощное средство программирования, которое позволяет определять набор методов, выполняемых как единое целое, что во многих случаях помогает улучшить структуру кода.

И напоследок, упомянем об ещё одной очень важной особенности использования делегатов. В программировании часто необходимо создать фрагмент кода (процедуру, метод), который будет «настраиваться» при конкретных применениях. Например, создать метод вывода на экран таблицы значений функции **f(x)**, математическое описание которой (конкретный вид) заранее неизвестно. При вызове такого метода в качестве фактического параметра может быть задана ссылка на программную реализацию конкретной функции **f(x)**. Так как делегат – это тип, экземпляры которого представляют ссылки на методы с заданной сигнатурой, то делегаты дают возможность рассматривать методы как сущности, ссылки на которые можно присваивать переменным соответствующего типа и передавать в качестве параметров. Таким образом, в метод можно передавать не только различные данные, но и различные процедуры (методы) их обработки. Такая функциональная параметризация применяется для создания универсальных методов и обеспечения возможности обратного вызова. Обратный вызов (**callback**) – это обращение из исполняемой процедуры к другой процедуре, которая часто определяется не при компиляции, а при выполнении программы, то есть обратный вызов – это вызов процедуры, которая передается в другую процедуру в качестве параметра. Таким образом, с использованием обратного вызова создаются методы, которые при своем исполнении вызывают другие (вспомогательные) методы, определённые зачастую не на этапе компиляции, а в точке вызова, т.е. в ходе выполнения проекта. Пример создания универсального метода с использованием делегата и обратного вызова рассмотрим в п. 4.5 Лабораторной работы.

### 4.2. События – члены класса

Событие – это член класса, позволяющий классу или объекту класса послать другим объектам сообщение об изменении своего состояния, или получении сообщения от других объектов.

В Windows система обработки сообщений (событий) – это фундаментальная структура, используемая для распространения информации в многозадачной среде. С точки зрения приложения сообщение является уведомлением о произошедшем событии, на которое программа должна (или не должна) каким-то образом отреагировать. Все создаваемые в среде .NET Framework приложения являются объектно-ориентированными, и управление ими осуществляется путем обработки сообщений от объектов – т.е. обработки событий. Большую часть времени Windows-приложения заняты обработкой событий пользова­теля, взаимодействующего с графическим интерфейсом. Когда от пользователя, запустившего приложение, поступает сообщение, это является свидетельством того, что он хочет изменить ход выполнения программы. Например, щелчок на кнопке сохранения файла означает, что пользователь решил запустить процесс сохранения информации в файле.

Итак, хотя проект Visual Basic можно представить, как набор процедур, выполняемых в определенной последовательности, на самом деле большинство программ управляются событиями, то есть поток выполнения определяется внешними воздействиями, называемыми событиями. Под событием в данном случае подразумевается сигнал, сообщающий приложению, что произошло нечто важное. Например, когда пользователь щелкает элемент управления на форме, форма может инициировать событие **Click** и вызывать процедуру, обрабатывающую событие. Или, например, пользовательское приложение выполняет задачу по сортировке отдельно от основного приложения. Если пользователь отменяет сортировку, то приложение может послать событие отмены, которое остановит процесс сортировки.

Поскольку сущность объектно-ориентированного программирования, в конечном счете, сводится к обмену сообщениями между взаимодействующими объектами, события должны занимать определенное место в этой схеме: объект-источник отправляет сообщение, которое указывает на возникновение события. Но при этом возникает очевидная проблема, каким объектам следует отправлять сообщения? Оповещать о каждом событии все объекты, существующие в настоящий момент в программе? Это было бы слишком неэффективно. Для большинства объектов событие не представляет ни малейшего интереса, а быстродействие станет неприемлемо низким. Вместо этого VB.NET пытается ограничить число получателей события, для чего используется модель «публикация/подписка». В этой модели объекты-приемники событий регистрируют объекты-источники тех событий, которые представляют для них интерес. Т.е. один класс, являющийся отправителем (**sender**) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся получателями (**receivers**) сообщения, подписываются на получение этих сообщений. Любой объект, способный вызывать события, является отправителем событий, также называемым источником событий. Формы, элементы управления и пользовательские объекты являются примерами отправителей событий. На события от одного источника могут подписаться сразу несколько объектов - приемников. О том, что источник инициировал событие, оповещаются только зарегистрированные получатели.

Общий смысл происходящего заключается в том, что при возникновении события объект-источник вызывает заранее определенные методы (процедуры) объектов-приемников, называемые обработчиками событий. Вызываемая процедура приемника регистрируется источником события одновременно с регистрацией объекта-приемника. Такая схема называется оповещением посредством обратного вызова (callback notification), потому что источник события вызывает метод приемника по заранее известному ему адресу.

Модель программирования, управляемая событиями и основанная на применении объектов, повышает производительность труда программиста. Стоило вам перетащить элемент на форму, как он начинал реагировать на определенные события. Например, код процедуры события **Button1\_Click()** выполнялся при нажатии кнопки с именем **Button1**. При этом заготовка этой процедуры – обработчика события нажатия на кнопку автоматически генерируется при размещении кнопки на форме:

|  |
| --- |
| **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **End Sub** |

Здесь процедуре-обработчику события (методу объекта-приёмника) передается два параметра:

* переменная **sender** имеет тип **Object** и указывает на объект, с которым произошло событие (содержит ссылку на объект - источник события, то есть кнопку, нажатую пользователем);
* переменная e указывает на объект класса, производного от **System.EventArgs**, который содержит (по крайней мере, теоретически) дополнительную информацию о событии (разные классы, производные от **System.EventArgs**, обладают разными свойствами, ориентированными на разные обработчики событий).

В данном случае объект события **е** не представляет интереса, поскольку он не содержит сколько-нибудь полезной информации о событии. С другой стороны, в некоторых ситуациях он может пригодиться. Например, из объектной переменной класса **MouseEventArgs** можно узнать, в какой точке был сделан щелчок мышью. В общем случае программист определяет собственные классы событий, производные от класса **System.EventArgs** и содержащие любую полезную информацию о событии.

Обратите внимание на ключевое слово **Handles** в определении процедуры события. Это ключевое слово переводится как «обрабатывает» и указывает, какие события обрабатываются данной процедурой. В данном примере процедура «обрабатывает событие **Click** объекта **Button1**. Наличие слова **Handles** заставляет процедуру запускаться в тот самый момент, когда возникает событие **Button1.Click**. В принципе одна процедура может обрабатывать несколько событий, для чего в конце объявления процедуры после **Handles** через запятую перечисляются эти события, правда только те, для обработки которых подходят параметры именно тех типов, которые указаны в скобках. Если стереть слова **Handles Button1.Click**, то обработчик события превратится в обычную процедуру пользователя и на события реагировать не будет.

Таким образом, если процедура имеет правильный набор параметров и в ее заголовке присутствует ключевое слово **Handles**, эта процедура может использоваться для обработки событий, например:

|  |
| --- |
| **Private Sub MyProc (ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **…**  **End Sub** |

Процедура **MyProc()** может обрабатывать событие **Button1.Click** благодаря наличию правильных параметров. Она обрабатывает это событие, поскольку в заголовке указано ключевое слово **Handles**. Другой пример, в котором одна процедура обрабатывает события сразу от двух разных кнопок и меню:

|  |
| --- |
| **Private Sub MyProc (ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click, \_**  **Button2.Click, Me.MenuComplete**  **…**  **End Sub** |

Таким образом, обработчики событий – это процедуры, вызываемые при возникновении соответствующего события. В качестве обработчика событий можно использовать любые допустимые подпрограммы с соответствующей сигнатурой. Функция, однако, не может быть использована в качестве обработчика событий, так как возвращение значения в источник события с ее помощью невозможно.

Мы рассмотрели пример события, вызванного пользователем (щелчком по кнопке) и обработчика этого события, причем взяли и то, и другое из готового набора, предложенного VB для элементов управления. Рассмотрим теперь примеры создания и обработки событий в собственных (пользовательских) классах.

В VB.NET события являются членами класса и должны быть объявлены при его определении. Как уже говорилось, класс, являющийся отправителем (источником) сообщения, публикует события, которые он может инициировать. Рассылка сообщения называется созданием события. Создание события в классе состоит из двух частей: описание события с помощью ключевого слова **Event,** задающее сигнатуру обработчиков событий, и описание метода (методов), инициирующих (вызывающих) событие. Для вызова события нужно применить оператор **RaiseEvent**. События должны вызываться в рамках класса, модуля или структуры, где они объявлены. Например, производный класс не может вызывать события, унаследованные из базового класса.

Обработка событий выполняется в классах – получателях сообщения. Для этого в них описываются методы – обработчики событий, сигнатура которых соответствует описанию события в классе – отправителе. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте - отправителе этот метод.

**Пример 4.2-1. Создать модель оплаты покупки в магазине кредитной картой, открытой на счёт клиента в банке. При совершении покупки со счета клиента в банке списывается задаваемая кассиром магазина сумма и для кассира печатается сообщение, что все прошло успешно. Если же сумма покупки превышает сумму денег на счёте клиента, то покупка отменяется, и кассиру приходит сообщение, что денег не хватает, а банк информируется о ненадёжности клиента.**

|  |
| --- |
| *'Класс - источник(отправитель) сообщений(событий)*  **Public Class СЧЁТ**  *'Закрытое поле -сумма денег на счете*  **Private S As Decimal = 200000**  *'Описание члена класса: событие при отрицательном балансе*  **Public Event Net()**  *'Описание члена класса: событие при любом снятии денег*  **Public Event Sob(ByVal z As String)**  *'Метод снятия денег со счета, инициирующий события*  **Public Sub Decrease(ByVal s1 As Decimal)**  **If S - s1 >= 0 Then**  **S = S - s1**  *'Оператор, порождающий событие Sob*  **RaiseEvent Sob("всё прошло успешно")**  **Else**  **RaiseEvent Sob("не хватает денег")**  *'Оператор, порождающий событие Net*  **RaiseEvent Net()**  **End If**  **End Sub**  **End Class** |

*Рис. 4.2-1. Программный код класса - источника сообщений* **СЧЁТ**

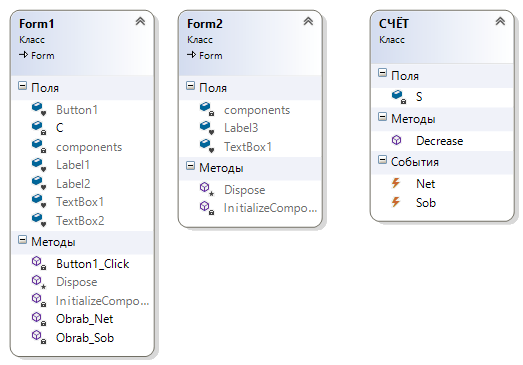
Проект состоит из класса и двух форм. Класс **СЧЁТ** моделирует счёт клиента в банке. Членами класса являются: закрытое поле **S** для хранения информации о сумме денег на счёте, два события, возникающие при попытке снять деньги со счёта, и метод снятия денег со счета, который уменьшает счет клиента на сумму покупки (если возможно), а также и инициирует события. Событие **Sob** имеет параметр – строку, в которую записывается сообщение для кассира об успешности или невозможности покупки. Событие **Net** не имеет параметров и возникает при нехватке денег на счёте клиента и предназначено банку. Программный код класса **СЧЁТ**, который является отправителем (источником) сообщений приведён на   
рис. 4.2-1.

Итак, в классе **СЧЁТ** создаются два события. Рассмотрим теперь методы их обработки. В этом примере первая форма моделирует магазин и имеет два текстовых поля (в первое поле вводится сумма покупки, во втором выводится сообщение для кассира об успешности или невозможности покупки), и кнопку. Вторая форма показывается при нехватке денег у покупателя, т.е. при возникновении события **Net** и имеет только одно текстовое поле, в которое записывается предупреждение для банка о ненадежности клиента. Таким образом, первая форма является классом - получателем сообщений и в этой форме будут описываться методы – обработчики событий, а вторая форма не содержит никакого программного кода.

|  |
| --- |
| *'Класс - получатель сообщения (с обработчиками событий)*  **Public Class Form1**  *'Создание объекта класса с событями это объект класса-источника*  **Private WithEvents C As New СЧЁТ**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim sum As Decimal** *' сумма для снятия со счета*  **sum = Val(TextBox1.Text)**  **C.Decrease(sum)** *'вызов метода снятия денег со счета*  **End Sub**  *'Обработчик события попытки снять деньги при любом балансе*  *' это реакция на событие**Sob источника - сообщение кассиру*  **Private Sub Obrab\_Sob(ByVal z As String) Handles C.Sob**  **TextBox2.Text = z**  **End Sub**  *'Обработчик события "нет денег" при отрицательном балансе*  *' это реакция на событие Net источника*  *'сообщение для банка в текстовое поле на второй форме*  **Private Sub Obrab\_Net() Handles C.Net**  **Form2.TextBox1.Text = "ненадежный клиент"**  **Form2.Show()**  **End Sub**  **End Class** |

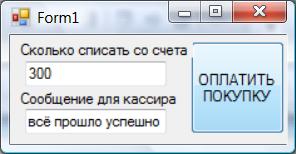
*Рис. 4.2-2. Программный код класса формы*   
*с обработчиками событий (первый способ)*

Для связи событий с их обработчиками, т.е. для обработки событий в VB имеется два способа. Первый способ самый простой и короткий: на уровне модуля (или класса) объявляется переменная (объект) класса – источника с ключевым словом **WithEvents** (т.е. объект класса с событиями). Тогда методы – обработчики событий в конце заголовка имеют ключевое слово **Handles**, после которого указано обрабатываемое событие. Программный код первой формы (класса **Form1**), которая является получателем сообщений и, соответственно, содержит их обработчики, приведён на рис. 4.2-2.

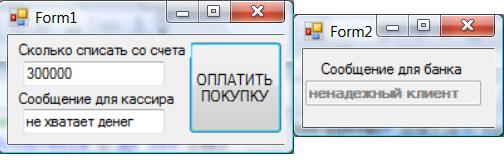


*Рис. 4.2-3. Схема классов с обработчиками событий (первый способ)*

Результаты работы проекта представлены на рис. 4.2-4 и рис. 4.2-5.



*Рис. 4.2-4. Результаты работы проекта при состоявшейся покупке*



*Рис. 4.2-5. Результаты работы проекта при нехватке денег на счете*

Рассмотрим теперь второй способ связывания событий с их обработчиками. Visual Basic позволяет динамически связывать обработчики событий с событиями путем создания делегата (т.е. посредством обратного вызова), используя оператор **AddHandler** («добавить обработчик»). Этот оператор регистрирует обработчик события и имеет два параметра: имя обрабатываемого события в классе – источнике и адрес метода (обработчика события) вызываемого при возникновении события в классе-приемнике. Синтаксис регистрации обработчика выглядит следующим образом:

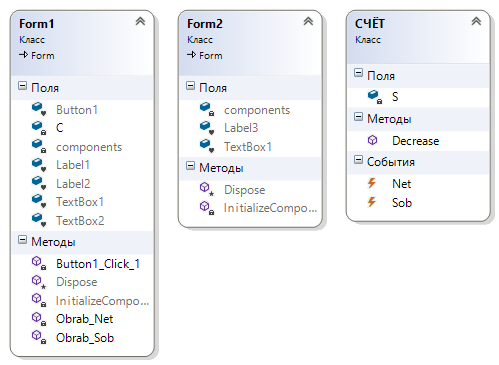
|  |
| --- |
| **AddHandler** *объект.событие***, AddressOf** *обработчик* |

В случае использования такого способа связывания события и обработчика, объявление объекта класса - источника со словом **WithEvents** и методов - обработчиков со словом **Handles** не применяется. Программный код первой формы (класса **Form1**), в которой регистрация обработчиков событий производится вторым способом, приведён на рис. 4.2-6, а Схема класса на рис. 4.2-7.

|  |
| --- |
| *'Класс - получатель сообщения (с обработчиками событий)*  **Public Class Form1**  *'создание объекта класса-источника*  **Private C As New СЧЁТ**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  *'Добавление обработчиков событий, т.е.обработчики событий*  *'связываются с событиями,*  *'вызванными объектом С класса-источника*  **AddHandler C.Sob, AddressOf Obrab\_Sob**  **AddHandler C.Net, AddressOf Obrab\_Net**  **Dim sum As Decimal** *' сумма для снятия со счета*  **sum = Val(TextBox1.Text)**  **C.Decrease(sum)** *'вызов метода снятия денег со счета*  **End Sub**  *'Обработчик события попытки снять деньги при любом балансе*  *' это реакция на событие источника - сообщение кассиру*  **Private Sub Obrab\_Sob(ByVal z As String)**  **TextBox2.Text = z**  **End Sub**  *'Обработчик события "нет денег" при отрицательном балансе*  *' это реакция на событие источника - сообщение банку*  **Private Sub Obrab\_Net()**  **Form2.TextBox1.Text = "ненадежный клиент"**  **Form2.Show()**  **End Sub**  **End Class** |

*Рис. 4.2-6. Программный код класса формы*

*с обработчиками событий (второй способ)*



*Рис. 4.2-7. Схема классов*

*с обработчиками событий (второй способ)*

Результаты работы этого приложения, использующего второй способ, полностью совпадают с представленными ранее на рис. 4.2-4 и 4.2-5 результатами первого способа.

Оба способа связывания событий с их обработчиками имеют свои преимущества и недостатки. Применение оператора **WithEvents** и предложения **Handles** (т.е. первый способ) часто является лучшим вариантом для обработчиков событий, поскольку используемый ими декларативный синтаксис делает обработку события проще для кодирования, чтения и отладки. Переменные **WithEvents** позволяют одному обработчику событий обрабатывать один или несколько видов событий или нескольким обработчикам событий обрабатывать один вид событий. При использовании первого способа очень удобно несколько событий обрабатывать одним обработчиком. Рассмотрим пример, в котором три события обрабатываются одной процедурой этим способом.

**Пример 4.2-2. Изменить разработанный в Примере 4.2-1 класс СЧЁТ, добавив в него метод пополнения счёта и событие, которое вызывается при добавлении денег, а также и свойство только для чтения, позволяющее клиенту узнать сумму денег, хранящуюся на счёте.**

Теперь класс содержит три события (**Dob, Spis** и **Net**), которые вызываются при пополнении счёта, при списании денег и при попытке снять со счёта сумму, превышающую ту, которая на нём хранится (т.е. при попытке получить отрицательный баланс). Все три события одинаковой сигнатуры и имеют в качестве формального параметра строку, в которую записывается сообщение для клиента об операции с его счётом.

|  |
| --- |
| **Public Class СЧЁТ** *'класс – источник*  *'Закрытое поле -сумма денег на счете*  **Private FS As Decimal = 200000**  *' Событие при отрицательном балансе*  **Public Event Net(ByVal z As String)**  *'Событие при пополнении счета*  **Public Event Dob(ByVal z As String)**  *'Событие при списании денег*  **Public Event Spis(ByVal z As String)**  *'Метод снятия денег со счета, инициирующий события*  **Public Sub Decrease(ByVal s1 As Decimal)**  **If FS - s1 >= 0 Then**  **FS = FS - s1**  **RaiseEvent Spis("Со счета списано" & s1 & " рублей")**  **Else**  **RaiseEvent Net("Нельзя списать " & s1 & " рублей")**  **End If**  **End Sub**  *'Метод добавления денег на счет, инициирующий событие*  **Public Sub Increase(ByVal s1 As Decimal)**  **FS = FS + s1**  **RaiseEvent Dob("Cчет пополнен на " & s1 & " рублей")**  **End Sub**  *'Св-во для чтения суммы денег на счету*  **Public ReadOnly Property S() As Decimal**  **Get**  **Return FS**  **End Get**  **End Property**  **End Class** |

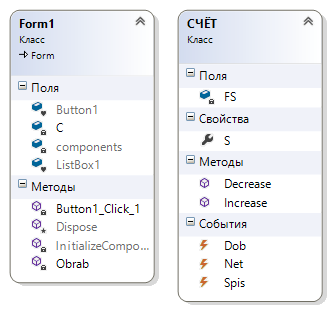
*Рис. 4.2-8. Программный код класса* ***СЧЁТ****,  
 объявляющий три события с одинаковой сигнатурой*

Проект этого примера состоит из изменённого класса **СЧЁТ** и одной формы. На форме есть элемент управления **ListBox**, в который для клиента выводятся сведения о состоянии его счёта и об операциях с ним. Программный код первой формы приведён на рис. 4.2-9.

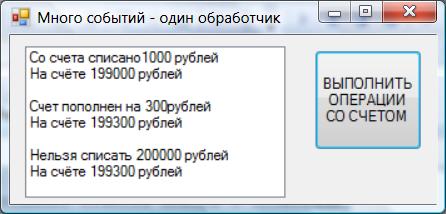
|  |
| --- |
| **Public Class Form1**  *'создание объекта класса с событиями*  **Private WithEvents C As New СЧЁТ**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **C.Decrease(1000)** *'вызов метода снятия денег со счета*  **C.Increase(300)** *'вызов метода добавления денег на счет*  **C.Decrease(200000)** *'попытка снятия слишком большой суммы*  **End Sub**  *'один обработчик на три события*  **Private Sub Obrab(ByVal z As String)Handles C.Net, C.Dob, C.Spis**  **ListBox1.Items.Add(z)**  **ListBox1.Items.Add("На счёте " & C.S & " рублей")**  **ListBox1.Items.Add("")**  **End Sub**  **End Class** |

*Рис. 4.2-9. Программный код класса* *формы*   
*с одним обработчиком трёх событий*

Класс первой формы объявляет объект C класса **СЧЁТ** как переменную с событиями (**WithEvents**), и содержит один обработчик (процедуру **Obrab()**) для всех трёх событий, которые вызываются этим объектом. Процедура **Obrab()** имеет в конце ключевое слово **Handles**, после которого перечисляются эти события, и производит вывод в **ListBox** сведений для клиента. В процедуре **Button1\_Click()** вызываются по очереди методы с заданным фактическим параметром для снятия и добавления денег. Вызовы этих методов инициируют три события, которые обрабатываются одной процедурой **Obrab().** Схема классов и результаты приведены на рис. 4.2-10 и 4.2-11.



*Рис. 4.2-10. Схема классов   
с одним обработчиком трёх событий*



*Рис. 4.2-11. Результаты работы программы*   
*с одним обработчиком для трёх событий*

Приведённый пример демонстрирует простоту и удобство обработки событий с применением ключевых слов **WithEvents** и **Handles**. Однако при использовании переменных **WithEvents** необходимо помнить о следующих ограничениях:

* нельзя использовать переменную **WithEvents** как объектную переменную. То есть ее невозможно объявить, как **Object** – при объявлении переменной необходимо указать имя класса;
* поскольку общие (т.е. статические – **Shared**) события не связываются с конкретными экземплярами класса, нельзя использовать **WithEvents** для обработки общих событий;
* нельзя также использовать **WithEvents** или **Handles** для обработки событий из структур (**Structure**) и невозможно создать массивы переменных **WithEvents**;
* основной проблемой синтаксиса **WithEvents** является его недостаточная гибкость. Хотя слово **Handles** является стандартным способом связывания события с обработчиком, связь посредством него может осуществляться только во время компиляции. Обработчики событий нельзя динамически (то есть во время выполнения программы) устанавливать и отключать на программном уровне – фактически вся схема обработки событий жестко фиксируется в программе.

Этих недостатков нет при втором способе связывания событий с обработчиками с помощью оператора **AddHandler**, который следует рассмотреть подробнее. Из синтаксиса этого оператора видно, что в качестве второго параметра в нём используется делегат, имя которого и является обработчиком события. Особенно эффективно применение делегатов в случаях, когда надо добавлять и удалять обработчики не на этапе компиляции, а во время выполнения программы. Например, необходимо, чтобы объект, инициирующий события, мог вызывать различные обработчики событий при различных условиях. Однако объект, вызывающий события, не может заранее знать время, когда обработчик событий обрабатывает конкретное событие. Использование оператора **AddHandler** позволяет динамически связывать обработчики событий с событиями путем создания делегата, и во время выполнения делегат передает вызовы соответствующему обработчику событий.

Хотя пользователь может создавать свои собственные делегаты, в большинстве случаев VB создает делегат в соответствии с имеющимися условиями. Например, оператор **Event** неявно определяет класс делегата с именем **EventHandler** как вложенный класс класса, содержащего инструкцию Event и с той же сигнатурой, которую имеет событие. Оператор **AddressOf** неявно создает экземпляр делегата, который ссылается на конкретную процедуру.

Ниже приведены две равнозначные строки кода. В первой строке показано явное создание экземпляра **EventHandle**r со ссылкой на метод **Button1\_Click()**, которая передается в качестве аргумента. Во второй строке представлен более удобный способ получения того же результата.

|  |
| --- |
| **AddHandler Button1.Click, New EventHandler(AddressOf Button1\_Click)** **AddHandler Button1.Click,** **AddressOf Me.Button1\_Click** |

Удаление (отключение) обработчика событий во время выполнения программы производится оператором **RemoveHandler**, который имеет синтаксис, аналогичный оператору **AddHandler**.

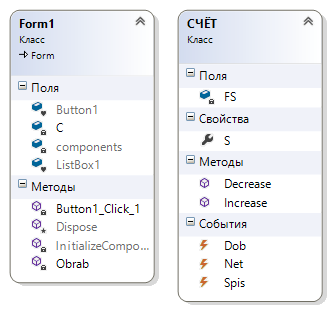
**Пример 4.2-3. Изменить код процедуры Button1\_Click() предыдущего Примера 4.2-2, используя второй способ связывания трёх событий класса СЧЁТ с обработчиком с помощью оператора AddHandler, добавляя и удаляя обработчик для конкретного события во время выполнения программы.**

Программный код класса **СЧЁТ** остался неизменным и приведён на рис. 4.2-8, программный код модуля формы приведён на рис. 4.2-12, а схема класса на 4.2-13.

В процедуре **Button1\_Click()** сначала обработчик (процедура **Obrab()**) связывается со всеми тремя событиями класса и по очереди вызываются методы, инициирующие эти события. Затем происходит отключение обработчика от события **Spis**, а потом и от события **Dob**, что для наглядности отражается в элементе управления **ListBox** первой формы. При этом повторяется вызов трёх методов класса, которые инициируют эти события. Таким образом, клиент будет информирован не обо всех операциях с его счётом. Результаты показаны на рис. 4.2-14.

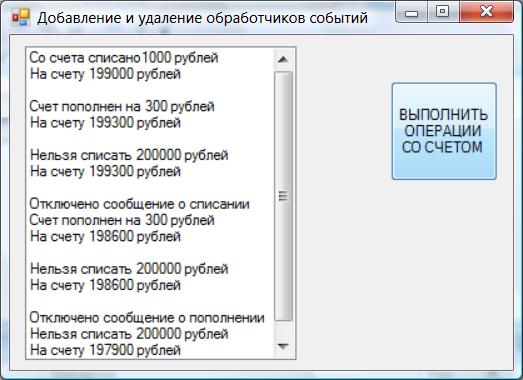
|  |
| --- |
| **Public Class Form1**  *'создание объекта класса-источника*  **Private C As New СЧЁТ**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  *'Добавление обработчика событий в список,*  *'т.е.обработчик связывается с событиями,*  *'вызванным объектом С класса-источника*  **AddHandler C.Net, AddressOf Obrab**  **AddHandler C.Dob, AddressOf Obrab**  **AddHandler C.Spis, AddressOf Obrab**  **C.Decrease(1000)** *'вызов метода снятия денег со счета*  **C.Increase(300)** *'вызов метода добавления денег на счет*  **C.Decrease(200000)** *'попытка снятия слишком большой суммы*  *'удаление события C.Spis из списка обработчика*  **RemoveHandler C.Spis, AddressOf Obrab**  **ListBox1.Items.Add("Отключено сообщение о списании")**  **C.Decrease(1000)** *'вызов метода снятия денег со счета*  **C.Increase(300)** *'вызов метода добавления денег на счет*  **C.Decrease(200000)** *'попытка снятия слишком большой суммы*  *'удаление события C.Dob из списка обработчика*  **RemoveHandler C.Dob, AddressOf Obrab**  **ListBox1.Items.Add("Отключено сообщение о пополнении")**  **C.Decrease(1000)** *'вызов метода снятия денег со счета*  **C.Increase(300)** *'вызов метода добавления денег на счет*  **C.Decrease(200000)** *'попытка снятия слишком большой суммы*  **End Sub**  *'Обработчик событий*  **Private Sub Obrab(ByVal z As String)**  **ListBox1.Items.Add(z)**  **ListBox1.Items.Add("На счету " & C.S & " рублей")**  **ListBox1.Items.Add("")**  **End Sub**  **End Class** |

*Рис. 4.2-12. Программный код класса формы* *с динамическим добавлением и удалением обработчика событий*



*Рис. 4.2-13. Схема классов с динамическим* *добавлением   
и удалением обработчика событий*

Таким образом, операторы **AddHandler и RemoveHandler** обладают большой гибкостью. Используя их, можно подключать и отключать события от обработчиков в зависимости от истинности или ложности каких-то условий, определяемых логикой программы, т.е. когда уведомления о событиях требуются только в определённых случаях, или, когда используются несколько обработчиков и нужный определяется во время выполнения программы. Или, например, обработчик событий связывается с некоторым событием, событие происходит. Обработчик событий перехватывает событие и выводит сообщение. Затем первый обработчик событий удаляется и с этим же событием сопоставляется другой обработчик событий. Когда событие происходит во второй раз, выводится другое сообщение.



*Рис. 4.2-14. Результаты работы* *программы*

*с динамическим добавлением и отключением обработчика событий*

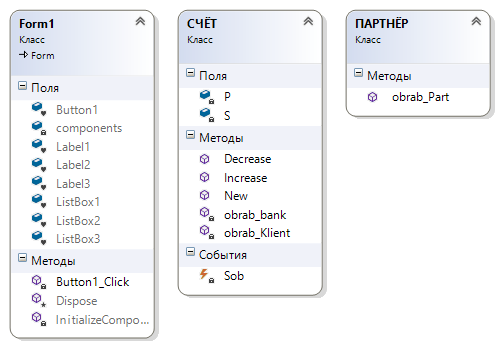
В рассмотренном примере 4.2-3 три события обрабатывались одним методом. Приведём пример, который демонстрирует ещё одно преимущество второго способа связи события с обработчиком оператором **AddHandler**. В нём одно закрытое событие обрабатывается несколькими обработчиками.

**Пример 4.2-4. В классе СЧЁТ объявить одно событие Sob, возникающее при любой операции со счётом и включить в него как закрытое поле ссылку на объект другого класса ПАРТНЁР, которому будет отправляться сообщение о возникновении события.**

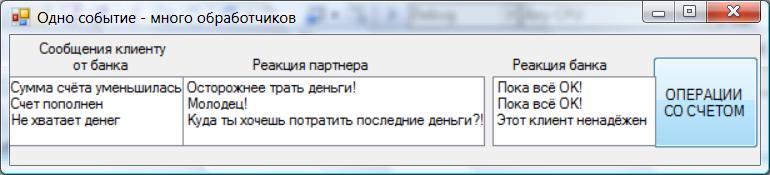
В учебных целях сделаем событие закрытым (**Private**) и в этом случае применение первого способа связывания события с обработчиком невозможно, так как закрытое событие будет недоступно извне класса и создание объекта этого класса с ключевым словом **WithEvents** не имеет смысла. Поэтому обработчики для этого события назначаются оператором **AddHandler**. Эти обработчики могут находиться и в классе **СЧЁТ**, и в других классах (правда, тогда они должны быть объявлены, как **Public**). В приведенном примере два закрытых обработчика (один с сообщением для клиента, другой – для банка) описываются в этом классе, а в классе **ПАРТНЁР** объявляется ещё один общедоступный обработчик для этого события – сообщение для партнёра (совладельца счёта) и его реакция на операции с общим счётом. Связь события с обработчиками устанавливается в конструкторе класса **СЧЁТ**. Так же, как и в предыдущем примере, в классе **СЧЁТ** имеются два метода: для пополнения счёта и для снятия денег. Для наглядности на форму выводятся результаты работы всех трёх обработчиков в три разных элемента управления **ListBox**. В процедуре **Button1\_Click()** модуля первой формы конструктором класса **СЧЁТ** создаётся локальный объект этого класса, и соответственно обработчики связываются с событием. Далее, по очереди вызываются методы, инициирующие событие **Sob** класса **СЧЁТ**. Программный код классов **СЧЁТ** и **ПАРТНЁР**, и также модуля формы приведён на рис. 4.2-15, схема класса на рис. 4.2-16, а результаты показаны на рис. 4.2-17.

|  |
| --- |
| **Public Class СЧЁТ** *'класс - источник*  *'Закрытое поле -сумма денег на счете*  **Private S As Decimal = 200000**  *'Закрытое поле – ссылка на объект класса ПАРТНЁР*  *'это композиция класса СЧЁТ и класса ПАРТНЁР*  **Private P As New ПАРТНЁР**  *'Описание закрытого события*  **Private Event Sob(ByVal z As String)**  *'В конструкторе класса происходит*  *'добавление обработчиков к событию*  **Public Sub New(ByVal z As String)**  **AddHandler Sob, AddressOf P.obrab\_Part**  **AddHandler Sob, AddressOf obrab\_bank**  **AddHandler Sob, AddressOf obrab\_Klient**  **End Sub**  *'Метод снятия денег со счета, инициирующий событие*  **Public Sub Decrease(ByVal s1 As Decimal)**  **If S - s1 >= 0 Then**  **S = S - s1**  **RaiseEvent Sob("Сумма счёта уменьшилась")**  **Else**  **RaiseEvent Sob("Не хватает денег")**  **End If**  **End Sub**  *'Метод добавления денег на счет, инициирующий событие*  **Public Sub Increase(ByVal s1 As Decimal)**  **S = S + s1**  **RaiseEvent Sob("Cчет пополнен")**  **End Sub**  *'Обработчик события для банка*  **Private Sub obrab\_bank(ByVal z As String)**  **If z = "Не хватает денег" Then**  **Form1.ListBox3.Items.Add("Этот клиент ненадёжен")**  **Else**  **Form1.ListBox3.Items.Add("Пока всё ОК!")**  **End If**  **End Sub**  *'Обработчик события для сообщения клиенту*  **Private Sub obrab\_Klient(ByVal z As String)**  **Form1.ListBox1.Items.Add(z)**  **End Sub**  **End Class**  *'Класс - получатель сообщения*  **Public Class ПАРТНЁР**  *'обработчик события у партнера*  *'реакция на событие источника*  **Public Sub obrab\_Part(ByVal z As String)**  **If z = "Сумма счёта уменьшилась" Then**  **Form1.ListBox2.Items.Add("Осторожнее трать деньги!")**  **ElseIf z = "Cчет пополнен" Then**  **Form1.ListBox2.Items.Add("Молодец!")**  **Else**  **Form1.ListBox2.Items.Add("Куда ты хочешь потратить" + \_**  **"последние деньги?!")**  **End If**  **End Sub**  **End Class**  **Public Class Form1**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim C As New СЧЁТ("")***'создание объекта класса-источника*  **C.Decrease(1000)** *'вызов метода снятия денег со счета*  **C.Increase(300)** *'вызов метода пополнения денег на счет*  **C.Decrease(200000)** *'попытка снятия слишком большой суммы*  **End Sub**  **End Class** |

*Рис. 4.2-15. Программный код классов* *для Примера 4.2-4*



*Рис. 4.2-16. Схема классов для Примера* *4.2-4*



*Рис. 4.2-17. Результаты работы* *программы*   
*с тремя обработчиками для одного события*

В этом примере все три обработчика активируются одновременно при создании объекта класса конструктором. Однако можно было активировать какой-то из них, например, обработчик для банка только в случае попытки снять со счёта слишком много денег. Тогда уведомление для банка о ненадёжности клиента возникало бы только при нехватке денег на счёте, и, следовательно, в конструктор, в метод класса **Decrease()** и в обработчик для банка необходимо внести следующие изменения (рис. 4.2-18):

|  |
| --- |
| **Public Class СЧЁТ** *'класс - источник*  **. . .**  *'В конструкторе класса происходит*  *'добавление только двух обработчика к событию*  **Public Sub New(ByVal z As String)**  **AddHandler Sob, AddressOf P.obrab\_Part**  **AddHandler Sob, AddressOf obrab\_Klient**  **End Sub**  *'Метод снятия денег со счета, инициирующий событие*  **Public Sub Decrease(ByVal s1 As Decimal)**  **If S - s1 >= 0 Then**  **S = S - s1**  **RaiseEvent Sob("Сумма счёта уменьшилась")**  **Else**  *'добавление обработчиков банка при нехватке денег*  **AddHandler Sob, AddressOf obrab\_ bank**  **RaiseEvent Sob("Не хватает денег")**  **End If**  **End Sub**  *'Обработчик события для банка*  **Private Sub obrab\_bank(ByVal z As String)**  **Form1.ListBox3.Items.Add("Этот клиент ненадёжен")**  **End Sub**  **. . .**  **End Class** |

*Рис. 4.2-18. Сокращённый программный код класса* ***СЧЁТ*** *с динамическим подключением обработчика события   
для банка только при нехватке денег*

И напоследок упомянем об обработке событий, унаследованных из базового класса. Производные классы наследуют все характеристики из базового класса, в том числе и события, поэтому производный класс может в любой момент инициировать открытые или защищенные события своего базового класса, при этом событие идентифицируется ключевым словом **MyBase**. Кроме того, производные классы автоматически наследуют все обработчики открытых и защищенных событий своих предков и могут обрабатывать события, инициированные их базовым классом. Однако часто в производном классе возникает необходимость в переопределении методов, используемых при обработке открытых и защищенных событий базового класса. Для этого при объявлении обработчика такого события в заголовок процедуры-обработчика производного класса добавляется конструкция

|  |
| --- |
| **Handles MyBase**.*имя\_события\_базового класса* |

Например:

|  |
| --- |
| *' Базовый класс*  **Public Class Class1**  *'Объявление события*  **Public Event BaseEvent(ByVal i As Integer)**  *'Далее где-то в классе объявлен обработчик этого события*  **. . .**  **End Class**  *'Производный класс*  **Public Class Class2**  **Inherits Class1**  *'Переопределение метода обработки события BaseEvent,*  *'унаследованного от базового класса*  **EventHandler(ByVal x As Integer) Handles MyBase.BaseEvent**  *'здесь пишется новый код обработчика события*  **End Sub**  **End Class** |

### 4.3. Обработка исключений в среде .NET Framework

При создании программ большое значение имеет обработка исключительных ситуаций (исключений) – ошибок, которые могут возникать при выполнении приложения и которые нельзя определить на этапе компиляции. Например, это может быть переполнение, деление на ноль, выход значения индекса массива за допустимые пределы, попытка открыть несуществующий файл, или сохранить информацию в базе данных на сервере при отсутствии соединения, и т. п.

Если в процессе работы программы возникла такая ошибка, система генерирует (выбрасывает) исключение, причём каждому типу ошибки соответствует своё исключение. В языке VB исключения – это объекты класса **System.Exception** или производных от него классов. Например, при переполнении будет сгенерировано исключение **OverflowException**, при недостатке памяти – исключение **OutOfMemoryException**. Обычно такие ситуации приводят к аварийному завершению программы с системным сообщением об ошибке.

В среде .NET Framework используется структурированный механизм обработки исключений, позволяющий восстановить работу программы без аварийного завершения и продолжить её выполнение. К основным преимуществам дан­ного механизма можно отнести использование единых средств поддержки обработки исключений во всех языках среды .NET, возможность создания защищенных блоков кода и фильтрации ис­ключений для эффективного управления ошибками, а также поддержку обработчиков исключе­ний, гарантирующих корректное завершение задач даже при возникновении фатальных ошибок. Среда .NET Framework предоставляет большое количество встроенных классов исключе­ний, используемых для обработки стандартных ошибок. Кроме того, среда .NET Frame­work позволяет программисту создавать собственные классы, которые используются для обработ­ки исключений, возникающих во время работы приложения.

Перехват и обработка исключений базируются на использовании конструкции **Try** . . . **Catch**. . . **Finally** . . . **End Try**, имеющей следующую структуру:

|  |
| --- |
| **Try**  [*Блок контроля*]  **Catch**  *[Блок обработчиков исключений*]  **Finally**  [*Завершающий блок*]  **End Try** |

В блок *контроля за исключениями*, который начинается после слова **Try**, помещают потенциально опасные операторы программы, т.е. операторы, которые могут вызвать генерацию исключения. Все процедуры прямо или косвенно вызываемые из блока, считаются принадлежащими блоку. Возникающее исключение перехватывается и обрабатывается кодом, помещенным в блок **Catch**. То есть после слова **Catch** программируется *обработчик исключений*, в котором записываются операторы, предназначенные для выполнения действий по обработке полученного исключения, причём обработчиков исключений (блоков **Catch**) может быть несколько для ошибок различных типов. Таким образом, операторы блока обработки исключений выполняются только в том случае, если в блоке контроля **Try** сгенерировалось исключение, которое соответствует утверждению (условию) этого блока **Catch**. С одним блоком **Try** может быть связано несколько блоков **Catch**. Каждый из этих блоков можно использовать для перехвата всех исключений, возникающих в блоке **Try**, или для обработ­ки исключений только определенного типа. Наличие нескольких блоков **Catch** позволяет динами­чески реагировать на исключения различных типов. Далее, за блоком (блоками) **Catch** может быть записан блок **Finally**, операторы которого выполняются независимо от того, было ли сгенерировано исключение в блоке контроля или нет. В этом блоке обычно записываются завершающие процедуры, такие как закрытие файлов, очистка буфера потока, освобождение объектов и т.п. Все эти блоки работают совместно, отсутствовать может либо блок **Catch** либо блок **Finally**, но не оба одновременно. Пример перехвата и обработки исключений, возникающих при чтении данных из файла, уже был рассмотрен в лабораторных работах Тем 1, 2, 3 (п.п. 1.5, 2.7, 3.6).

Наверняка вы сталкивались с исключительной ситуацией, когда пользователь вместо числа вводит последовательность нечисловых символов или вводит число в неправильном формате. В этом случае программа завершается аварийно и выдаёт сообщение о необработанном исключении **InvalidCastException**, которое выбрасывается при неверном преобразовании типа. Рассмотрим пример перехвата и обработки такого исключения.

**Пример 4.3-1. Написать программу, контролирующую правильность ввода значения вещественного числа.**

Приведённый на рис. 4.3-1 программный код содержит бесконечный цикл, который не завершится до тех пор, пока пользователь не введёт значение вещественного числа x без ошибок. Т.е. строка ввода числа не должна содержать нечисловых символов, кроме показателя степени (экспоненты, например, 1E10) и для отделения дробной части числа от целой части используется запятая, а не точка.

|  |
| --- |
| **Option Strict On**  **Public Class Form1**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim x As Double**  **Dim buf As String**  **Do** *'бесконечный цикл*  **Try** *'Блок контроля*  **buf = InputBox("введите число")**  **x = CDbl(buf)** *'возможно возникновение исключения*  **TextBox1.Text = "введено число " & CStr(x)**  **Catch ex As InvalidCastException** *'Блок обработки исключения*  **MsgBox("Ошибка в формате числа!")**  **Continue Do** *'продолжение цикла - переход к оператору Loop*  **End Try**  **Exit Do** *'выход из цикла*  **Loop**  **End Sub**  **End Class** |

*Рис. 4.3-1. Программный код программы, контролирующей правильность ввода значения вещественного числа*

Эта программа содержит бесконечный цикл, выход из которого возможен только при достижении конца тела цикла, где находится оператор **Exit Do**. В теле цикла - блок контроля за возникновением исключений **Try**, в котором записаны три оператора. Во втором операторе возможна генерация исключения. Если оно появляется, то третий оператор пропускается, а управление передается за пределы блока контроля, т.е. обработчику исключения - блоку **Catch**. В этом блоке два оператора: первый выдаёт сообщение об ошибке, второй – передает управление на следующую итерацию цикла (оператор **Loop**, минуя оператор **Exit Do**). Таким образом, выход из цикла возможен только тогда, когда пропускаются операторы блока обработки исключений **Catch**, т.е. когда пользователь ввёл число без ошибок и исключения не возникло. В этом случае выполняется третий оператор блока контроля, и безошибочно введённое число выводится в текстовое поле **TextBox1** на форме.

Итак, контроль за возникновением исключений и их обработкой реализуется следующим образом. Сначала выполняются операторы блока контроля **Try**. Функция или операция, в которой возникла ошибка, генерирует исключение. Если исключений не возникло, то все **Catch**-обработчики пропускаются, а иначе выполнение текущего блока **Try** прекращается и управление передаётся **Catch**-блокам, которые просматриваются до тех пор, пока не будет обнаружен обработчик, «настроенный» на переданное исключение. После выполнения операторов этого обработчика выполняется блок завершения **Finally**, если он есть, причем он выполняется и в том случае, если ошибки не возникло. Стоит отметить, что среди блоков обработки выполняется только один (или не выполняется ни одного).

Очень важно, что обработка исключения не прекращает выполнение программы. В **Catch**-блоках можно запрограммировать действия по устранению причин появления исключения, и программа продолжает выполняться.

Однако приведенная программа имеет существенный недостаток. Если пользователь введёт синтаксически правильное числовое значение, которое выходит из диапазона вещественных чисел, например, 1E999, то возникнет ситуация, при которой будет сгенерировано исключение **OverflowException**. Так как оно отличается от предусмотренного программой исключения **InvalidCastException**, то произойдёт аварийное завершение программы с выдачей сообщения о необработанном исключении **System.OverflowException**. Чтобы защитить программу от аварийного завершения при возникновении исключений такого типа, можно добавить в неё ещё один **Catch**-блок:

|  |
| --- |
| **. . .**  *'Блок обработки исключения при переполнении*  **Catch ex As OverflowException**  **MsgBox("Слишком большое или слишком маленькое число")**  **Continue Do**  **End Try** |

Более общим решением будет дополнение программы **Catch**-блоком, настроенным на перехват всех исключений типа **ArithmeticException**, относящихся к обработке арифметических данных или даже ещё более общий случай, который мы рассмотрим в следующем примере.

**Пример 4.3-2. Написать программу, использующую несколько блоков Catch для обработки исключений различных типов при работе с целыми числами и с массивом целых чисел.**

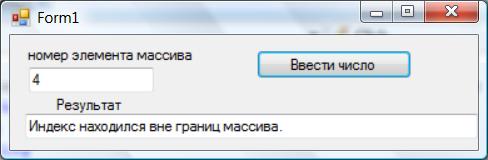
Условие фильтрации ошибок в каждом блоке **Catch** основывается на классе возникшего исключения. Так как каждое исключение – это объект класса **System.Exception** или его наследника, то при обработке исключений можно использовать свойства этого класса, например, очень удобное для пользователя свойство **Message**, которое содержит текстовое описание ситуации, при которой создано исключение, что и показано в приведённом на рис. 4.3-2 программном коде.

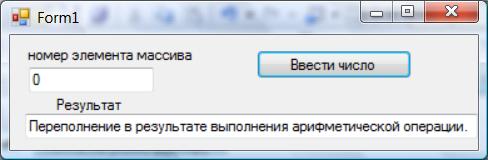
|  |
| --- |
| **Public Class Form1**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim a, b, k As Integer**  **Dim mas() As Int32 = {0, 1, 2, 3}**  **Dim s As String**  **Try** *'ввод целого числа – номера элемента массива*  **s = TextBox1.Text**  *'Приведение к числовому типу - возможна ошибка*  **k = Convert.ToInt32(s)**  *'Чтение элемента массива –возможен выход за пределы массива*  **a = mas(k)**  *'Деление - возможна арифметическая ошибка*  **b = 10 / a**  **TextBox2.Text = CStr(b)**  **Catch ex As IndexOutOfRangeException**  *'выход за пределы массива –*  *'в TextBox2 выводится значение*  *'свойства Message объекта ex*  **TextBox2.Text = ex.Message**  **Catch ex As ArithmeticException**  *'арифметическая ошибка*  **TextBox2.Text = ex.Message**  **Catch ex As Exception**  *'общая ошибка*  **TextBox2.Text = ex.Message**  **End Try**  **End Sub**  **End Class** |

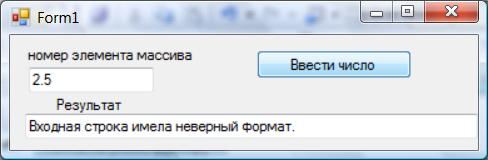
*Рис. 4.3-2. Код программы с несколькими обработчиками исключений,*

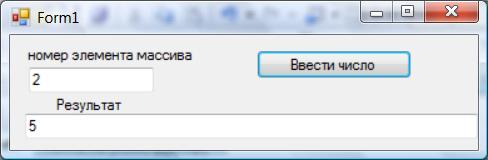
*в которых фильтрация ошибок основывается на классе*

*возникшего исключения*









*Рис. 4.3-3. Результаты работы программы   
с несколькими обработчиками исключений*

В этой программе в текстовое поле **TextBox1** вводится целое число, которое является индексом элемента массива и предусмотрены два конкретных обработчика ошибок, сгенерированных в первом случае при выходе из диапазона массива, т.е. если вводится число, большее, чем 3 (так как номер последнего элемента массива равен 3). Второй обработчик настроен на арифметическую ошибку, возникающую при делении целого числа на ноль (если вводится число 0). Третий блок перехватывает все остальные исключения (например, неправильный формат ввода целого числа). Очень важно помнить, что в **Catch**-блоках сначала надо писать частные исключения и в самом конце – общие, как показано в примере. Результаты работы программы с перехваченными ошибками трёх описанных типов, которые выводятся в **TextBox2** с использованием свойства **Message**, и при правильном вводе числа представлены на рис. 4.3-3.

Возможен и другой способ фильтрации ошибок – с помощью любого условного выражения, как показано в следующем примере.

**Пример 4.3-3. Написать программу, использующую блок Catch для обработки исключений с помощью условного выражения.**

|  |
| --- |
| **Public Class Form1**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim x As Integer = 5**  **Dim y As Integer = 0**  **Try**  **x = x / y**  **Catch ex As Exception When y = 0**  **MsgBox(ex.Message)**  **End Try**  **End Sub**  **End Class** |

*Рис. 4.3-4. Код программы с обработчиком исключений   
на основе условного выражения*

Программный код для перехвата ошибки при y=0 приведён   
на рис. 4.3-4.

Мы рассмотрели исключения от библиотечного класса исключений **System.Exception** или его наследников. Встроен­ные классы обрабатывают большинство типов ошибок. Однако программист может сам сгенерировать исключение из нужной точки программы. Если при выполнении программы возникает исключение, которое не принад­лежит ни одному из встроенных классов, можно сформировать собственные исключения, выпол­няющие обработку ошибок в пользовательском коде. Исключение как объект создаётся с помощью специального оператора генерации (посылки) исключения **Throw**. Очень часто так решается задача проверки корректности исходных данных, получаемых процедурой.

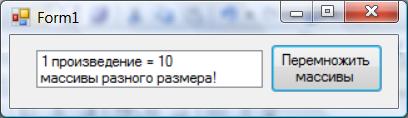
**Пример 4.3-4. Написать процедуру вычисления скалярного произведения двух одномерных массивов. Если массивы имеют разное число элементов, сгенерировать собственное исключение с помощью оператора Throw.**

Программный код для перехвата ошибки в случае попытки перемножить массивы разного размера приведён на рис. 4.3-5.

Как видно из этого кода, внутри функции **proiz()**, которая вычисляет произведение двух массивов, выполняется проверка на равенство размерностей массивов. Если размеры массивов не равны, необходимо определить собственное исключение, чтобы не вычислять некорректное произведение, что и выполняется внутри условного оператора **If**. При генерировании исключения конструктором **New** создается объект класса **System.Exception**. Строка-параметр конструктора ("**массивы разного размера!"**) задаёт значение свойства **Message**. В процедуре **Button1\_Click()** задаются три массива из 4 чисел и один – из трёх чисел. Вызовы метода **proiz()** размещены в **Try**-блоке. Первый вызов корректен, во втором допущена ошибка – размеры входных массивов не совпадают. Результаты работы этой программы приведены на рис. 4.3-6.

|  |
| --- |
| **Imports System.Math**  **Public Class Form1**  *'Ф-ция вычисления скалярного произведения двух массивов*  **Function proiz(ByVal x() As Double, ByVal y() As Double) As Double**  *' Проверка равенства размерности массивов*  **If x.Length <> y.Length Then**  *'Генерация собств. исключения*  **Throw New Exception("массивы разного размера!")**  **End If**  **Dim i As Integer**  **Dim P As Double = 0**  *'Вычисление произведения в случае отсутствия ошибки*  **For i = 0 To x.Length - 1**  **P = P + x(i) \* y(i)**  **Next**  **P = Sqrt(P)**  **Return P**  **End Function**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  *'Объявление массивов*  **Dim x1() As Double = {1, 3, 5, 7}**  **Dim y1() As Double = {2, 4, 6, 8}**  **Dim x2() As Double = {1, 2, 3, 4}**  **Dim y2() As Double = {1, 2, 3}**  **Dim P1, P2 As Double**  **Try**  **P1 = proiz(x1, y1)** *'1 вызов ф-ции - ошибки нет*  **ListBox1.Items.Add("1 произведение = " + CStr(P1))**  **P1 = proiz(x2, y2)** *'2 вызов ф-ции - будет ошибка*  **ListBox1.Items.Add("2 произведение = " + CStr(P2))**  **Catch ex As Exception** *'обработка ошибки*  **ListBox1.Items.Add(ex.Message)**  **End Try**  **End Sub**  **End Class** |

*Рис. 4.3-5. Код программы вычисления скалярного произведения  
 двух массивов с генерацией пользовательского исключения.*



*Рис. 4.3-6. Результаты работы* *программы*

*с собственным обработчиком* *исключений*

Иногда в программе возникает ошибка, которую можно исправить с помощью исключения, а затем продолжить выполнение оставшейся части кода блока **Try**. Например, если произошла ошибка, связанная с попыткой деления на нуль, результату можно присвоить нулевое значение, вывести сообщение об ошибке и продолжить выполнение данной программы. Реализуется это пу­тем использования нескольких вложенных блоков **Try**. В таком случае код, при выполнении кото­рого может возникнуть ошибка, должен находиться во внешних блоках **Try** и **Catch**. После обработ­ки исключения во внутреннем блоке **Try**, управление вновь должно быть передано внешнему блоку **Try**. На рис. 4.3-7 приведен фрагмент кода, где обрабатывается исключение, которое не перехватывается внутренним блоком Try, но зато перехватывается внешним блоком **Try.**

|  |
| --- |
| **. . .**  **Try**  **Try**  **X1 = X1 / X2**  **Catch e As DivideByZeroException**  **X1 = 0**  **End Try**  *'Продолжение выполнения кода*  **Catch**  *'Код обработки внешнего исключения*  **End Try**  **. . .** |

*Рис. 4.3-7. Фрагмент кода программы с вложенными блоками* ***Try***

### 4.4. Задачи для самостоятельного решения

Решить задачи из **Темы 2** (п. 2.6, задание 2) с обработкой исключений. Методы и конструкторы, реализуемые в классе, должны выполнять проверку передаваемых параметров или получаемых значений, в затем генерировать исключения в случае возникновения ошибок, используя подходящие стандартные исключения или исключения – наследники от стандартных исключений, и/или определить собственные исключения.

### 4.5. Лабораторная работа по теме

### «*Делегаты, события и обработка исключений*»

**Цель** данной лабораторной работы состоит в изучении основных понятий объектно-ориентированной технологии таких, как делегаты, события и обработка исключений и получении практических навыков в применении этих средств при разработке приложений, а также закреплении полученных при изучении **Темы 3** сведений о виртуальных методах и абстрактных классах.

#### 4.5.1. Задание

1. ***Изучите средства описания и использования*** *делегатов, событий и обработку исключений в VB (п.п. 4.1– 4.3), а также повторите средства описания и использования абстрактных классов и виртуальных методов (п.п. 3.2, 3.3).*
2. ***Изучите общее задание*** *в п. 4.5-2 и выберите вариант индивидуального задания из табл. 4.5-1.*
3. ***Для каждого проекта******разработайте концептуальную модель*** *задачи, то есть:*

* проведите объектно-ориентированную декомпозицию,определив множество образующих ее классов*;*
* определите для выделенных классов существенные в данной задаче интерфейсные члены класса*:*
* *атрибуты;*
* *операции (возможные действия) объектов, которые объекты могут выполнить для решения задачи.*

1. ***Разработайте объектные модели*** *задачи логического уровня, определив необходимые связи между классами объектов, в том числе иерархию классов, для чего:*

* определите базовый(е) класс(ы) и их структуру*;*
* определите классы – наследники*;*
* создайте схему (иерархию) наследования классо*в.*

1. ***Проведите дальнейшую формализацию****, для чего:*

* определите перечень исходных данных*;*
* представьте смысловые данные, которые будут использоваться в ходе решения задачи в виде переменных или массивов, присвоив им соответствующие имена;
* определите размерности и типы используемых данных*;*
* при необходимости приведите расчетные формулы, которые используйте для преобразования исходных данных в результатыигеометрическую иллюстрацию заданного метода исследования функции*.*

1. ***Разработайте интерфейс пользователя – две формы в каждом проекте:***

* *форму, предназначенную для ввода исходных данных;*
* *форму вывода таблиц значений различных функций на заданных интервалах с заданным шагом и вывода результатов заданных методов исследования.*

1. ***Создайте программный код для каждого проекта, реализующий поставленную задачу и получите объектную модель на уровне реализации.***
2. ***Выполните проект и получите решение.***
3. ***Обоснуйте правильность полученных результатов.***

#### 4.5.2. Варианты индивидуальных заданий

Создать два проекта в одном решении. В каждом проекте реализовать универсальные метод табулирования и метод исследования функции на заданном диапазоне изменения аргумента от **a** до **b** с заданным числом равноотстоящих точек (т.е. с вычисляемым шагом **h**). При вводе исходных данных провести контроль правильности ввода с выбрасыванием исключения в случае некорректного ввода. В первом проекте использовать делегат как параметр методов табулирования и исследования, во втором проекте применить переопределение виртуального метода абстрактного класса в наследниках. Построить таблицы значений и исследовать две различные функции на различных диапазонах с различным числом равноотстоящих точек. Варианты методов исследования функций приведены в табл. 4.5-1

Таблица 4.5-1

|  |  |
| --- | --- |
| 1)  2)  3)  4)  5)  6)  7)  8)  9)  10)  11)  12)  13)  14)  15)  16)  17)  18)  19)  20)  21)  22)  23)  24)  25)  26)  27)  28)  29)  30) | Вычислить количество точек графика функции, лежащих внутри окружности с заданным радиусом **r** и центром в заданной точке **(xc,yc)**.  Вычислить количество точек графика функции, попадающих в область, ограниченную равнобедренным прямоугольным треугольником с катетами заданной длины **L** и пересекающимися в заданной точке **(xc,yc)**.  Вычислить количество значений функции, совпадающих с заданным значением **y1** с абсолютной погрешностью **ε**.  Определить границы первого отрезка длиной **h** (шага изменения аргумента функции), на концах которого функция имеет разные знаки.  Вычислить количество точек на графике функции, лежащих левее прямой, проходящей через заданную точку (**xc,0**) с углом наклона 45º.  Вычислить координаты точки первого экстремума функции.  Вычислить количество точек на графике функции, попадающих в 1-ый и 3-ий квадранты координатной плоскости.  Вычислить количество точек графика функции, попадающих в область, ограниченную первым полупериодом синусоиды и осью абсцисс.  Вычислить сумму и количество значений функции, принадлежащих заданному интервалу [**y1;y2**].  Вычислить минимальное по абсолютной величине значение функции и соответствующее значение аргумента.  Вычислить количество точек на графике функции, попадающих в прямоугольник, заданный координатами левого верхнего и правого нижнего углов.  Вычислить количество точек на графике функции, попадающих внутрь сектора окружности с радиусом **r** и центром в заданной точке (**xc,yc**). Одна сторона сектора параллельна оси абсцисс, угол сектора - 45º.  Вычислить количество точек на графике функции, лежащих левее прямой, проходящей через начало координат с углом наклона 135º.  Вычислить количество точек на графике функции, лежащих вне кольца с радиусами **r1** и **r2** и центром в заданной точке (**xc,yc**).  Вычислить количество точек на графике функции, принадлежащих области, ограниченной параболой **y=cx2** и прямой **y=d (d>0)**.  Вычислить количество интервалов длиной **h** (шага изменения аргумента функции), на которых функция изменяет знак.  Вычислить среднее геометрическое положительных значений функции.  Вычислить количество точек графика функции, лежащих внутри кольца с радиусами **r1** и **r2** и центром в заданной точке (**xc,yc)**.  Вычислить количество точек на графике функции, попадающих в квадрат с заданными координатами левого нижнего угла **(c,d)** и длиной стороны **L**.  Вычислить количество точек на графике функции, лежащих правее прямой, проходящей через начало координат с углом наклона 45º.  Вычислить количество точек на графике функции, попадающих во 2-ой и 4-й квадранты координатной плоскости.  Вычислить произведение и количество значений функции, лежащих вне заданного интервала **[y1;y2].**  Определить границы последнего отрезка длиной h(шага изменения аргумента функции), на концах которого функция имеет разные знаки.  Вычислить максимальное по абсолютной величине значение функции и соответствующее значение аргумента.  Вычислить количество точек на графике функции, лежащих правее прямой, проходящей через заданную точку **(0,yc)** с углом наклона 45º.  Вычислить минимальное из положительных значений функции и соответствующее значение аргумента.  Вычислить максимальное из отрицательных значений функции и соответствующее значение аргумента.  Вычислить количество точек на графике функции, попадающих в квадрат с заданными координатами правого верхнего угла **(c,d)** и длиной стороны **L**.  Вычислить количество точек на графике функции, попадающих в треугольник, образованный прямыми, проходящими через точки с координатами **(c,c),  (-c,c)** и (**0,0**).  Вычислить среднее арифметическое отрицательных значений функции. |

#### 4.5.3. Содержание отчета

1. Тема и название работы.
2. Общее задание на разработку моделей и программных проектов, и вариант индивидуального задания.
3. Концептуальные модели предметной области задачи.
4. Объектные модели программных проектов на логическом уровне и (при необходимости) геометрическая иллюстрация метода исследования.
5. Графический интерфейс пользователя каждого проекта:

форма, предназначенная для ввода исходных данных;

форма для вывода результатов решения задачи.

1. Программные проекты на уровне реализации (приложения) с подробными комментариями:
2. Содержание классов проектов;
3. Схемы алгоритмов функциональных задач;
4. Программный код проектов.
5. Схемы объектных моделей, построенных по программному коду.
6. Результаты выполнения приложения.
7. Обоснование правильности полученных результатов.

#### 4.5.4. Пример выполнения задания

1. ***Тема и название работы:***

Делегаты, события и обработка исключений.

Универсальный метод вывода таблицы значений функции на заданном отрезке и с заданным числом равноотстоящих точек таблицы двумя способами: с использованием делегатов; с помощью абстрактного класса и его наследников.

1. ***Общее задание на разработку моделей и программного проекта, и вариант индивидуального задания.***

Создать универсальный метод вывода таблицы значений функции на заданном отрезке с заданным числом равноотстоящих точек таблицы двумя способами: с использованием делегатов; с помощью абстрактного класса и его наследников. Требуется разработать для каждого способа отдельный проект (два проекта в одном решении).

1. ***Концептуальная модель предметной области и задачи.***
2. **Проведите объектно-ориентированную декомпозицию, определив множество образующих классов.**

В результате анализа предметной области задачи и объектно-ориентированной декомпозиции определен образующий класс **Form1** **(рис. 4.6-1 а)** дляпервого проекта**,** а для второго проекта– образующий абстрактный класс **ТАБЛИЦА** **(рис. 4.6-1 б).**



а) б)

Рис. 4.5-1. *Объектные модели программных проектов   
на концептуальном уровне*

1. **Определите для выделенных классов существенные в данной задаче интерфейсные члены класса:**

* **Входные атрибуты**:

для рассматриваемой задачи нет.

* **Выходные атрибуты**:

для рассматриваемой задачи нет.

* **Операции:**
* ***построение таблицы значений функции*** на заданном отрезке с заданным числом равноотстоящих точек;
* ***метод(ы), задающие вид вычисляемой функции***.

Объектная модель программного проекта на концептуальном уровне с учетом интерфейсных членов класса представлена на рис. 4.5-2 а) для первого случая, а для второго случая на рис. 4.5-2 б).



*Рис. 4.5-2*. *Объектные модели программных проектов   
на концептуальном уровне с учетом интерфейсных членов класса*

1. ***Объектная модель программного проекта на логическом уровне:***
   1. **Уточните состав классов объектной модели, проведите дальнейшую формализацию, уточните логическую структуру и связи между классами.**

При анализе предметной области на логическом уровне, заметим, что для первого проекта в классе формы требуется объявить вложенный тип–делегат, задающий сигнатуру методов вычисления функций, которые вызываются с его помощью.

Во втором проекте описывается абстрактный класс **ТАБЛИЦА** с виртуальным методом вычисления функции и обычным методом построения таблицы значений этой функции. Так как в задаче требуется построить таблицу для двух различных функций, то переопределение виртуального метода (задание конкретного вида функции) происходит в двух классах – наследниках **Class1** и **Class2**.

В универсальный метод **tablica()** как параметры передаются диапазон значений аргумента [a;b] и количество равноотстоящих точек **n**. В первом проекте **ДЕЛЕГАТ** в универсальный метод **tablica()** также в качестве параметра передаётся вид вычисляемой функции через функцию-делегат для обеспечения обратного вызова. Во втором проекте **НАСЛЕДНИК** универсальный метод **tablica()** объявляется в абстрактном классе и использует виртуальный абстрактный метод описания функции, а конкретный вид вычисляемой функции переопределяется в классах – наследниках. В общем задании для студентов требуется ввод исходных данных (**a, b** и **n**) для двух разных функций производить на первую форму с контролем правильности ввода (Пример 4.3-1), а вывод результатов произвести на вторую форму каждого проекта.

В рассматриваемом примере в каждом проекте используется только одна форма. С помощью универсального метода вывести на форму в элемент управления **TextBox1** таблицу из 10-ти значений функции **sin(x)** на отрезке **[0;π],** а в **TextBox2** – таблицу10-ти значений функции **x2** на отрезке **[1;5]**.

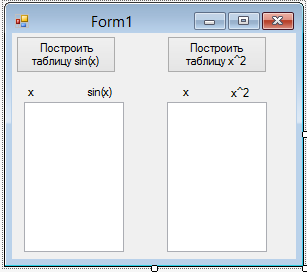
Таким образом, на логическом уровне объектные модели можно представить на рис. 4.5-3

а) б)

*Рис. 4.5-3.**Объектные модели на логическом уровне*

1. ***Графический интерфейс пользователя:***

*Форма интерфейса пользователя приведена на рис. 4.5-4.*

**

*Рис. 4.5-4. Форма для построения таблиц значений двух* *функций*

1. ***Объектная модель задачи на уровне реализации:***

6.1) Содержание классов для двух проектов:

Определим содержание полученных проектов и их классов:

**Проект ДЕЛЕГАТ:**

**Класс Form1:**

Поля: нет

Свойства для чтения и записи полей класса: нет

* Методы:
* две процедуры – функции **f1()** и **f2()** задающие конкретный вид для двух вычисляемых функций;
* процедура **tablica()** построения таблицы значений функции на заданном отрезке с заданным числом равноотстоящих точек (значения функции собираются в одну строку);
* две событийные процедуры **Button1\_Click()** и **Button2\_Click()** вывода таблиц значений для двух различных функций в элементы управления **TextBox1** и **TextBox2** соответственно.
* Вложенный тип – делегат:

|  |
| --- |
| **Delegate Function Func(ByVal x As Double) As Double** |

**Проект ТАБЛИЦА:**

**Базовый абстрактный класс ТАБЛИЦА:**

Поля: нет

Свойства для чтения и записи полей класса: нет

* Методы:
* абстрактная процедура – функция **f()**, которая будет переопределяться в классах - наследниках;
* процедура **tablica()** построения таблицы значений функции на заданном отрезке с заданным числом равноотстоящих точек (значения функции собираются в одну строку).

**Производный класс Class1 – наследник класса ТАБЛИЦА:**

* ***Методы***:
* переопределенный (**Overrides**) метод базового класса **f()**- процедура-функция, задающая конкретный вид вычисляемой функции (возвращающая значение **Sin(x)**).

**Производный класс Class2 – наследник класса ТАБЛИЦА:**

* ***Методы***:
* переопределенный (**Overrides**) метод базового класса **f()**- процедура-функция, задающая конкретный вид вычисляемой функции (возвращающая значение **x^2**).

**Класс Form1:**

* ***Методы:***
* две событийные процедуры **Button1\_Click()** и **Button2\_Click()** вывода таблиц значений для двух различных функций в элементы управления **TextBox1** и **TextBox2** соответственно.

**6.2. Схема алгоритма метода исследования:**

Схемы алгоритмов метода исследования студенты выполняют самостоятелно.

**6.3) Программные коды проектов приведены на рис 4.5-5 и 4.5-6.**

|  |
| --- |
| **Imports System.Math**  **Public Class Form1**  *'Объявление делегата - типа*  **Public Delegate Function Func(ByVal x As Double) As Double**  *'Объявление функций для построения таблиц*  **Public Function f1(ByVal x As Double) As Double**  **Return Sin(x)**  **End Function**  **Public Function f2(ByVal x As Double) As Double**  **Return x ^ 2**  **End Function**  *'Определение универсального метода построения таблицы*  *'из n точек значений функции от одного аргумента*  *'на отрезке [a; b].*  *'Параметр по ссылке z* – *строка, в которую «собирается»*  *'таблица для последующего вывода в TextBox*  **Public Sub tablica(ByRef f As Func, ByVal a As Double, \_**  **ByVal b As Double, ByVal n As Integer, \_**  **ByRef z As String)**  **Dim x, y, h As Double**  **Dim i As Integer**  **z = ""**  **h = (b - a) / n** *'шаг изменения x*  **x = a**  **For i = 0 To n**  **y = f(x)**  **z = z + Format(x, "0.00") + Space(5) + \_**  **Format(y, "0.00") + vbNewLine**  **x = x + h**  **Next**  **End Sub**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim z As String = ""** *'строка для вывода*  *'вызов метода построения таблицы из 10 точек*  *'для функции f1=sin(x) на отрезке [0; PI]*  **tablica(AddressOf f1, 0, PI, 10, z)**  **TextBox1.Text = z**  **End Sub**  **Private Sub Button2\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button2.Click**  **Dim z As String = ""**  *'вызов метода построения таблицы из 10 точек*  *'для функции f2=x^2 на отрезке [1; 5]*  **tablica(AddressOf f2, 1, 5, 10, z)**  **TextBox2.Text = z**  **End Sub**  **End Class** |

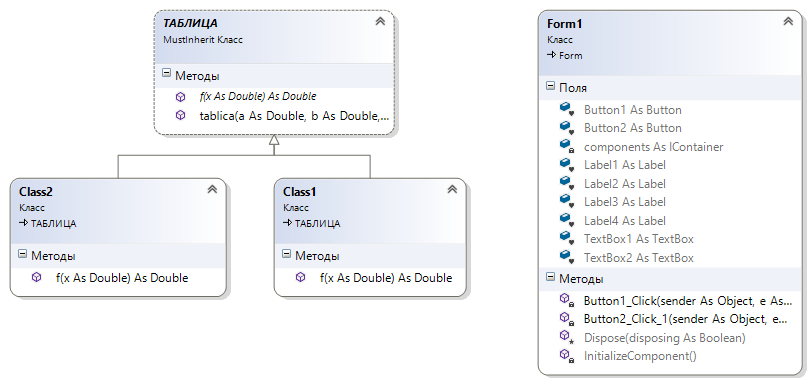
*Рис. 4.5-5. Программный код проекта* ***Делегат***   
*построение таблиц значений двух функций   
с использованием делегата в качестве параметра метода*

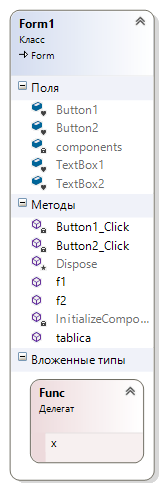
|  |
| --- |
| *'модуль формы*  **Imports System.Math**  **Public Class Form1**  **Private Sub Button1\_Click(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button1.Click**  **Dim z As String = ""** *'строка для вывода*  **Dim R As New Class1** *'создание объекта класса*  *'вызов метода для объекта R - таблицы sin(x):*  **R.tablica(0, PI, 10, z)**  **TextBox1.Text = z**  **End Sub**  **Private Sub Button2\_Click\_1(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs) \_**  **Handles Button2.Click**  **Dim z As String = ""**  **Dim G As New Class2**  *'вызов метода для объекта G - таблицы x^2:*  **G.tablica(1, 5, 10, z)**  **TextBox2.Text = z**  **End Sub**  **End Class**  *'Базовый абстрактный класс*  **Public MustInherit Class ТАБЛИЦА**  *'абстрактный метод - функция для вывода таблицы*  **Public MustOverride Function f(ByVal x As Double) As Double**  *' метод вывода таблицы из n значений функции*  *' на отрезке[a; b] в строку z*  **Public Sub tablica(ByVal a As Double, ByVal b As Double, \_**  **ByVal n As Integer, ByRef z As String)**  **Dim x, y, h As Double**  **Dim i As Integer**  **z = ""**  **h = (b - a) / n** *'шаг изменения x*  **x = a**  **For i = 0 To n**  **y = f(x)**  **z = z + Format(x, "0.00") + Space(10) \_**  **+ Format(y, "0.00") + vbNewLine**  **x = x + h**  **Next**  **End Sub**  **End Class**  **Imports System.Math**  *'1класс-наследник*  **Public Class Class1**  **Inherits ТАБЛИЦА**  *'переопределение абстракного метода конкретной ф-цией sin(x)*  **Public Overrides Function f(ByVal x As Double) As Double**  **Return Sin(x)**  **End Function**  **End Class**  *'2класс-наследник*  **Public Class Class2**  **Inherits ТАБЛИЦА**  *'переопределение абстракного метода конкретной ф-цией x^2*  **Public Overrides Function f(ByVal x As Double) As Double**  **Return x ^ 2**  **End Function**  **End Class** |

*Рис. 4.5-6. Программный код проекта* ***Наследник*** - *построение   
таблиц значений двух функций   
с помощью абстрактного класса и его наследников*

1. ***Схемы классов объектных моделей уровня реализации, построенных по программному коду:***

Схемы классов проектов уровня реализации, построенных по программному коду, приведены на рис. 4.5-7 а) и 4.5-7 б)*.*



 а)

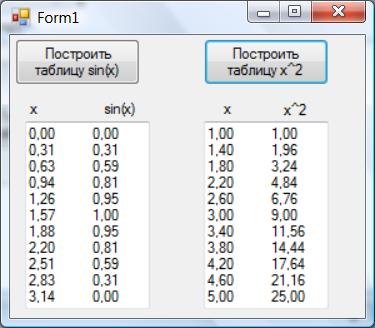
*Рис. 4.5-7. Схемы классов уровня реализации*

1. *Объектная модель проекта* ***Делегат***  
   *построение таблиц значений двух функций   
   с использованием делегата в качестве параметра метода*

б) *Объектная модель проекта* ***Наследник***   
*построение таблиц значений двух функций   
с помощью абстрактного класса и его наследников*

1. ***Результаты выполнения приложения.***

Результаты работ проектов совпадают и приведены на рис. 4.5-8.



*Рис. 4.5-8. Результаты выполнения проектов* ***Делегат*** *и* ***Наследник***

### 4.6. Контрольные вопросы

1. Что такое делегат? Для чего нужны делегаты?
2. Как описать делегат-тип?
3. Как создать экземпляр делегата?
4. Опишите порядок создания и вызова многоадресного делегата.
5. Для чего нужны групповые делегаты?
6. Можно ли сформировать групповой делегат из методов-функций?
7. Что такое обратный вызов? Когда он используется?
8. Как можно использовать делегат для создания универсальных методов? Каким образом можно создать универсальный метод без использования делегата?
9. Каким образом объекты посылают друг другу сообщения?
10. Как описать член класса - событие? Каким образом генерируется событие?
11. Для чего используются модель «подписка/публикация»?
12. Что является обработчиком события?
13. Какими способами можно связать событие с обработчиком в языке VB? Назовите преимущества и недостатки каждого способа.
14. Можно ли одно событие обработать несколькими обработчиками и несколько событий – одним обработчиком? Если да, то как? Если нет – то почему?
15. Что такое динамическое подключение и отключение обработчиков событий? Как его осуществить?
16. Что такое исключение? Перечислите известные вам стандартные исключения.
17. Как осуществляется обработка исключений в языке VB?
18. Каким образом можно сгенерировать собственное исключение?
19. Из чего состоит блок обработки исключений? Опишите порядок работы этого блока.
20. Каким образом можно продолжить выполнение программы при возникновении исключительной ситуации?