МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

В.М. Коновалов

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**Учебно-методическое пособие
по проведению практических занятий**

*для студентов IV курса
направления 01.03.04
очной формы обучения*

**Москва
2017**

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра прикладной математики
В.М. Коновалов

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие
по проведению практических занятий

для студентов IV курса
направления 01.03.04
очной формы обучения

Москва-2017

ББК 6Ф7.3
К63

Рецензент канд. техн. наук, доц. Н.И. Овсянникова

Коновалов В.М.

К63 Проектирование программного обеспечения: учебно-методическое пособие по проведению практических занятий. – М.: МГТУ ГА, 2017. – 36 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Проектирование программного обеспечения» по учебному плану для студентов IV курса направления 01.03.04 очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 28.02.2017 г. и методического совета 28.02.2017 г.

Подписано в печать 02.03.2017 г.

Печать офсетная	Формат 60x84/16	1,59 уч.-изд. л.
2,08 усл.печ.л.	Заказ № 1725/150	Тираж 40 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д.20
ООО «ИПП «ИНСОФТ»
107140, г. Москва, 3-й Красносельский переулок д.21, стр.1

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

В содержание методического пособия отобран учебный материал, который может оказать помощь студентам при выполнении индивидуальных курсовых заданий. Одной из целей курсового проектирования является разработка программных моделей информационных систем, максимально приближенных к условиям эксплуатации. Это означает, что разрабатываемое приложение должно адекватно реагировать на различные события, приводящие к **сбоям** в работе кода приложения **на этапе выполнения**.

Современные языковые средства разработки приложений и поддерживающие их работу инструментальные среды содержат **средства**, позволяющие, в определенной степени, оградить пользователя от ошибок выполнения (**run-time errors).** Это относится к офисным приложениям, выполненным на VBA в среде MS Office, так и разработанным на языках группы .NET в инструментальной среде разработки Visual Studio.

В первом разделе пособия представлены сведения об обработке ошибок времени выполнения, поясняющие принципы перехвата и обработки ошибок (исключений). Рассмотрение детализировано до уровня объяснения синтаксических конструкций блоков обработки, встраиваемых в код приложения. Приводятся примеры кода обработчиков ошибок для **типовых** ситуаций, на основе изучения которых можно строить процедуры обработки ошибок выполнения для любой индивидуальной ситуации, приводящей к **run-time error.**

На конкретных примерах иллюстрируется преимущество написания **структурированных** обработчиков ошибок выполнения. Обращается внимание на различие технологий защитного программирования и процесса обработки ошибок, исключающего обрушение приложения, и даже делающим незаметным для пользователя (в ряде случаев) возникновение ошибочной ситуации.

Разрабатываемые студентами в ходе курсового проектирования приложения реализуют одну из выбираемых технологий (**DAO, ADO, ADO .NET**) доступа к внешним источникам данных. Поэтому приложения носят характер **приложений баз данных**. По этой причине, во втором разделе представлен материал справочного типа, используемый при написании кода приложения, связанного с обращением к данным в базе.

Конструирование нетривиальных запросов к **СУБД,** реализующих требуемую функциональность приложения в соответствии с курсовым заданием, основывается на понимании синтаксических особенностей **SQL**-операторов.

В содержание второго раздела пособия включены сведения, предоставляющие возможность написания достаточно сложных запросов, используя **стандартные SQL**-инструкции, реализуемые в любых диалектах **SQL**-языка современных серверов баз данных.

1. Обработка ошибок времени выполнения (run-time errors)

В VBA имеются развитые средства обнаружения и устранения ошибок. Их использование состоит в стремлении исключить ошибки при разработке приложения либо задать реакцию среды на ошибку при выполнении программы.
Ниже рассматривается процесс обработки ошибок (в средах VB **v.6**, VBA, VB.NET), возникающих **во время выполнения** программы (run-time error). Причинами таких ошибок, (также называемых исключениями) могут быть как ошибки в самой программе, так и обстоятельства, находящиеся вне сферы влияния разработчика. Например, отсутствие требуемых файлов в момент обращения к ним, отказы аппаратных средств или сети, а также некорректные действия пользователя.
Так или иначе, выполнение программы прерывается и выводится сообщение, в котором, обычно, мало полезной для пользователя информации: указывается недопустимая операция, а не причина возникновения ошибки. Самая привлекательная программа теряет свои достоинства, если она работает со сбоями.

В практике разработки приложений используют два подхода, позволяющие оградить (в определенной степени) программный продукт от ошибок выполнения:
- предотвращение возникновения ошибочных ситуаций;
- обработка ошибки с помощью специальной процедуры.

Первый подход характеризуется тем, что программно анализируются вводимые или вычисляемые данные. В том случае, если они могут приводить к ошибке, программа информирует пользователя о необходимости корректного представления данных.
При разработке программ рекомендуется, по возможности, предотвращать возникновение ошибочных ситуаций, отвечая на такие, к примеру, вопросы как: существует ли файл, который требуется открыть; находится ли курсор в требуемой позиции для возможности старта макроса и т. п.

Не все ошибки можно предотвратить, поэтому используют второй подход, характеризующийся тем, что если ошибка все-таки произошла, то она перехватывается, обрабатывается, то есть создается программная реакция на исключение.
При создании приложений необходимо сочетать оба подхода, применяя в каждом конкретном случае тот подход, который кажется разработчику наиболее эффективным.

1.1. Перехватываемые ошибки

Перехват прерывания по ошибке используется в том случае, когда предотвратить возникновение ошибочной ситуации невозможно. Ошибочная ситуация может сопровождаться, например, такими сообщениями:

Номер ошибки	Сообщение об ошибке по умолчанию
5	**Procedure call or argument is not valid** (Недопустимый вызов процедуры или аргумент)
6	**Overflow** (Переполнение)
7	**Out of memory** (Недостаточно памяти)
11	**Division by zero** (Деление на ноль)
13	**Type mismatch** (Несоответствие типов переменных)
51	**Internal error** (Внутренняя ошибка)
53	**File not found** (Файл не найден)
61	**Disk full** (Диск переполнен)
440	**Automation error** (Ошибка программирования)
462	**The remote server machine does not exist or is unavailable** (Удаленная серверная машина не существует или недоступна)
463	**Class not registered on local machine** (Класс не зарегистрирован на локальной машине)
...	

Для обычного пользователя (не профессионала) это – малозначащие сообщения. Предотвращая возникновение ошибок, или перехватывая их, можно исключить вывод стандартных сообщений. Вместо них следует отобразить либо более простую подсказку, либо попытаться исправить ошибочное состояние с помощью кода, обрабатывающего ошибку.

1.2. Общие сведения о перехвате ошибок

Обработка ошибок выполнения в VBA называется перехватом ошибок. Правильно разработанное приложение должно обеспечивать отклик на многие возможные ошибки. Но еще более важным является то, чтобы ошибка **не остановила** выполнение процедуры. При этом пользователь может даже не заметить, что произошла ошибка.

Общие правила перехвата ошибок можно рассмотреть на примере следующей процедуры – шаблона:

```
Sub MyProcedure ()
' . . .
    On Error GoTo MyErrorHandler
' . . .
' Инструкции, при выполнении каждой из которых
' может произойти ошибка
' . . .
    On Error GoTo 0
' . . .
Exit Sub   ' Обход подпрограммы обработки ошибки
MyErrorHandler:
```

```
'. . .
' Инструкции обработчика ошибки
'. . .
Resume
End Sub
```

Обработчик ошибок – это код для перехвата и обработки ошибок в приложении. Использование ваших **собственных** обработчиков ошибок говорит о том, что пользователи перестанут получать сообщения об ошибках на малопонятном языке сообщений Windows.

Для создания собственного обработчика ошибок необходимо выполнить следующие действия:

■ Установить ловушку прерываний (error trap) указанием места кода в процедуре, куда следует перейти при возникновении ошибки (адрес обработчика). Это действие осуществляется посредством оператора **On Error** , который делает доступным перехват прерываний и указывает метку, за которой начинается код обработчика прерываний.

■ Написать процедуры обработки прерываний, связанных с ошибками вашего приложения.

■ Обеспечить выход из процедуры обработки прерывания.

Рассмотрим более подробно каждый из трех вышеперечисленных этапов создания собственного обработчика ошибок.

Инструкция **On Error** устанавливает перехват для невыполненной части кода, указывая на подпрограмму обработки ошибки. В процедуре приложения может быть **несколько** операторов **On Error,** каждый из которых определяет разные обработчики ошибок.

Инструкция **On Error** имеет три синтаксические формы:

● **On Error GoTo метка** – позволяет передать управление подпрограмме обработки ошибки, которая идентифицируется меткой. Меткой является любое, допустимое в синтаксисе языка программирования, имя. Метка располагается в самом начале строки, а сразу за ней ставится двоеточие.

● **On Error Resume Next** – позволяет игнорировать ошибку и продолжить выполнение процедуры приложения со следующей инструкции – после той, при выполнении которой произошла ошибка. Если данная ошибка может привести к появлению других ошибок, то лучше не применять рассматриваемую форму инструкции **On Error.**

● **On Error GoTo 0** – останавливает перехват прерывания по ошибке в данной процедуре и, следовательно, отключает обработку ошибки. Эта форма инструкции **On Error** обычно указывается после первых двух форм – ниже строк кода, в которых могут возникнуть ошибки.

В том случае, если оператор **On Error GoTo 0** не используется, то ловушка остается активной на время работы процедуры (функции), в которой она установ-

лена, т.е. до того момента, пока не выполнится один из операторов: **Exit Sub, Exit Function, End Sub** или **End Function** этой процедуры (функции).

В каждый момент времени в данной процедуре может быть доступна только одна ловушка. Но можно создать несколько ловушек и активизировать их в разное время. В любом случае инструкция **On Error** устанавливает перехват и обработку ошибок только в той процедуре, в которой она указана.

При написании блока кода обработки прерывания по ошибке первым оператором является метка. Далее (в VBA), без всяких ограничивающих блоки кода операторов, следует писать код, который, вообще говоря, необязательно должен обрабатывать ошибки. Можно просто констатировать факт наличия ошибки. Это – не лучший способ написания обработчика. Но на это есть две причины: во-первых, не всегда можно идентифицировать ошибку, а во-вторых, не всякую ошибку можно исправить. В подобных случаях можно только предполагать, что же произошло на самом деле и дать пользователю рекомендательные сообщения о последующих действиях.

Перед меткой, задающей начало обработчика, должен находиться оператор выхода из процедуры (функции) – **Exit Sub (Exit Function).** В противном случае код обработчика будет выполняться даже при отсутствии ошибки, поскольку подпрограмма обработки ошибки является частью процедуры.

В приводимом ниже примере обработчик выполняется в том случае, если пользователь вводит данные не целого типа:

```
Private Sub Command1_Click()
   Dim z As Integer, y As Integer
   On Error GoTo ErrorHandler
   y = Val (InputBox("Введите целое значение", "Целочисленные вычисления"))
   z = y + 10
   MsgBox  "z =  "& z , ,"Результат"
   MsgBox "Спасибо!"
   Exit Sub
ErrorHandler:
   MsgBox ("Просили же Вас ввести целое значение!")
End Sub
```

Как видно из этого примера, здесь даже не делается попытка идентифицировать ошибку. В такой простой ситуации можно только предположить, что виной всему будет пользователь, который игнорирует все подсказки о необходимости ввода целого числа в диалоговом окне.

Вышеприведенный код можно дополнить анализом произошедшей ошибки. В Visual Basic есть возможность узнать тип ошибки посредством **объекта Err**, свойства которого Number и Description позволяют определить номер ошибки и **стандартное** описание ситуации в том виде, как ее оценивает сам Visual Basic.

С учетом этого, предыдущий код в обработчике можно усилить операторами анализа номера ошибки:

```
Private Sub Command1_Click()
   Dim z As Integer, y As Integer
   On Error GoTo ErrorHandler
   y = Val (InputBox("Введите целое значение", "Целочисленные вычисления"))
   z = y + 10
   MsgBox  "z =  "& z , ,"Результат"
   MsgBox "Спасибо!"
   Exit Sub
ErrorHandler:
   If err.Number = 13 Then
      MsgBox ("Просили же Вас ввести целое значение!")
   End If
End Sub
```

Очевидно, что рассмотренная процедура иллюстрирует лишь принцип перехвата ошибки. Понятно, что сразу после выполнения кода обработчика вся процедура завершается. Такой обработчик хорош только в качестве примера того, как не следует поступать. Если по всяким пустякам процедура будет завершать выполнение, то с таким приложением будет трудно работать. Необходим способ обработать ошибку или, **в крайнем случае**, сообщить о ней пользователю, но обязательно попытаться **продолжить** выполнение процедуры. От того, как реализован выход из блока обработки прерывания по ошибке, зависит дальнейшее развитие событий.

Для выхода из блока обработки ошибки можно вообще ничего не предпринимать и работа процедуры (или функции) будет завершена. Но поскольку с этой процедурой могут быть связаны другие части кода приложения, то просто отказаться от ее дальнейшего выполнения будет не всегда правильно. По крайней мере, надо как-то сообщить о том, что процедура завершилась из-за ошибки.

Выход из блока обработки ошибок происходит с помощью инструкции **Resume**, которая указывается в конце подпрограммы обработки ошибок. Она возобновляет исполнение процедуры, в которой произошла ошибка.

Существует три синтаксические формы инструкции **Resume:**

- **Resume** или **Resume 0** – передает управление инструкции, в которой возникла ошибка, и VBA производит попытку выполнить эту инструкцию вновь. При этом предполагается, что обработчик устранил **причины**, которые вызвали ошибку, и теперь можно вновь выполнить инструкцию без ошибки.
- **Resume Next** – передает управление инструкции, которая следует за той, в которой возникла ошибка. При этом предполагается, что обработчик устранил **последствия** ошибки, хотя причины ее возникновения могут остаться.

Примечание: Следует различать условия использования оператора **Resume Next** в конце подпрограммы обработки ошибок и в ловушке – **On Error Resume Next.**

● **Resume метка** – передает управление инструкции, идентифицированной указанной меткой

Пример использования инструкции **Resume** приведен ниже. Это самая простая форма оператора, при использовании которой происходит возврат к строке кода, в которой произошла ошибка. По крайней мере, в рассматриваемом случае этого вполне достаточно, поскольку после правильного ввода числа программа будет работать корректно:

```
Private Sub Command1_Click()
   Dim z As Integer, y As Integer
   On Error GoTo ErrorHandler
   y = Val (InputBox("Введите целое значение", "Целочисленные вычисления"))
   z = y + 10
   MsgBox  "z =  "& z , ,"Результат"
   MsgBox "Спасибо!"
   Exit Sub
ErrorHandler:
   If err.Number = 13 Then
      MsgBox ("Просили же Вас ввести целое значение!")
      Resume
   End If
End Sub
```

В следующем примере, в коде обработчика используется оператор **Resume Next.** Предполагается, что у пользователя возникли проблемы с вводом чисел, при этом обработчик предлагает пользователю свой вариант ввода. Затем код продолжает выполняться со следующей строки (за оператором **Val (InputBox( ))**:

```
Private Sub Command1_Click()
   Dim z As Integer, y As Integer
   On Error GoTo ErrorHandler
   y = Val (InputBox("Введите целое значение", "Целочисленные вычисления"))
   z = y + 10
   MsgBox  "z =  "& z , ,"Результат"
   MsgBox "Спасибо!"
   Exit Sub
ErrorHandler:
   If err.Number = 13 Then
      MsgBox ("С Вашего разрешения введем 9999")
      y = 9999
```

```
      Resume Next
   End If
End Sub
```

Следующий пример иллюстрирует использование обеих форм инструкции **Resume.** Обработчик дает пользователю возможность трехкратного ввода числа. Для этого используется как оператор **Resume,** так и **Resume Next.** В код процедуры введен счетчик **Counter**, который инициализируется при загрузке формы (процедура **Form_Load).** В блоке обработки ошибок значение счетчика увеличивается на единицу. Как только пользователь использует три попытки ввода числа, оператор **Resume Next** обработчика прекратит его действия, предлагая свой вариант ввода:

```
Public Counter As Integer
Private Sub Command1_Click()
   Dim z As Integer, y As Integer
   On Error GoTo ErrorHandler
   y = Val (InputBox("Введите целое значение", "Целочисленные вычисления "))
   z = y + 10
   MsgBox  "z =  "& z , ,"Результат"
   MsgBox "Спасибо!"
   Exit Sub
ErrorHandler:
   If err.Number = 13 Then
      If Counter = 2 Then
         MsgBox ("С Вашего разрешения введем 9999")
         y = 9999
         Resume Next
      Else
         Counter = Counter + 1
         MsgBox ("У вас осталось " &  (3 - Counter) & " попытки")
         Resume
      End If
   End If
End Sub

Private Sub  Form_Load ( )
   Counter = 0
End Sub
```

1.3. Объект Err и обработчики ошибок

Подпрограмма обработки ошибок должна выполнять определенные действия **в зависимости** от возникшей ошибки. С помощью объекта **Err,** как уже отмечалось выше, можно во время выполнения процедуры выявить **стандартный** тип ошибки. Объект **Err** содержит информацию о последней возникшей ошиб-

ке. Используя свойства объекта **Err** и выполняя его методы, можно сообщить обработчику характер произошедшей ошибки.

Объект **Err** имеет шесть свойств и два метода. Наиболее существенными для данного рассмотрения являются:

- Свойство **Number** – содержит номер возникшей ошибки.
- Свойство **Source** – хранит имя проекта VBA, в котором произошла ошибка.
- Свойство **Description** – это строка стандартного описания ошибки, соответствующая ее номеру. Например, для отображения описания ошибки можно использовать инструкцию:

```
MsgBox  Err.Number & " : " & Err.Description
```

Некоторые ошибки не имеют стандартного описания, и тогда строка имеет значение по умолчанию: "**Ошибка, определяемая приложением**" или "**Ошибка, определяемая объектом**".

Если предположить, что в процедуре могут произойти самые разнообразные ошибки, то чтобы произвести определенные действия в зависимости от типа возникшей ошибки, необходимо проверять значение свойства **Number.** Для этого используется любая логическая инструкция, например, **If-then-else-end If**, **Select Case ... end Select**. Последняя инструкция удобнее, так как в нее проще добавлять дополнительные условия, например:

```
ErrorHandler:
   Select Case  Err.Number
      Case  52, 53, 75, 76
          'Подпрограмма обработки ошибок
      Case  55
          'Другая подпрограмма обработки ошибок
      Case  71
          'Еще одна подпрограмма обработки ошибок
      Case Else
          'Подпрограмма обработки непредвиденных ошибок
   End Select
Resume
```

Использование в этом фрагменте инструкции **Case Else** также оправдано. Оно может служить целям информирования пользователя.

1.4. Примеры процедур, использующих перехват ошибок

В приводимых ниже примерах, иллюстрирующих различные ситуации, связанные с отсутствием доступа к требуемому источнику данных, обработчик ошибок может вызываться:

- если отсутствует диск в дисководе (**G** – к примеру);
- если не удается найти путь к требуемой папке (**G:\XLFiles**);
- если отсутствует требуемый файл **Book1.xls** в **G:\XLFiles**-папке (каталоге).

Типовые ошибки

```
Sub MyMacro ()
    Dim MyWorkbook As Workbook
  ' Установим перехват потенциально возможной ошибки.
    On Error GoTo Errhandler
    ChDrive "G:"
    ChDir "G:\"
    ChDir "G:\XLFiles"
    Workbooks.Open "Book1.xls"
  ' Остановим перехват ошибок, поскольку они не произошли в интересующих
  ' нас инструкциях.
    On Error GoTo 0
    Set MyWorkbook = ActiveWorkbook
    MsgBox "The destination workbook is " & MyWorkbook.Name
  ' Обходим обработчик ошибки, так как он не понадобится, и завершаем процедуру.
    Exit Sub
Errhandler:
  ' Если ошибка произошла, выдаем предупреждение пользователю и завершаем
  ' процедуру.
   MsgBox "An error has occurred. The macro will end."
End Sub
```

В этом примере используется оператор On Error для отображения сообщения и завершения процедуры при возникновении ошибки. Обработчик ошибок выводит следующее сообщение:

Произошла ошибка. Макрос будет завершен.

После этого выполнение процедуры завершается. Если книга **Book1.xls** успешно открыта, отображается сообщение, показывая целевую книгу, и макрос прекращает работу, так как имеется оператор **Exit Sub**, перед меткой обработчика ошибок – **ErrHandler**.

Генерация обработчиком специфического сообщения об ошибке

Данный пример похож на макрос в предыдущем примере. Однако этот макрос использует объект **Err** для того, чтобы вывести более детальное сообщение об ошибке при ее возникновении.

```
Sub MyMacro()
  Dim MyWorkbook As Workbook
  ' Run the Error handler "ErrHandler" when an error occurs.
  On Error GoTo Errhandler
  ChDrive "G:"
  ChDir "G:\"
```

```
  ChDir "G:\XLFiles"
  Workbooks.Open "Book1.xls"
  ' Disable the error handler.
  On Error GoTo 0
    Set MyWorkbook = ActiveWorkbook
    MsgBox "The destination workbook is " & MyWorkbook.Name
    ' Exit the macro so that the error handler is not executed.
    Exit Sub
Errhandler:
    Select Case Err.Number
        Case 68, 75:    ' Error 68: "Device not available"
                        ' Error 75: "Path/File Access Error"
            MsgBox "There is an error reading drive G."
        Case 76:        ' Error 76: "Path not found"
            MsgBox "The specified path is not found."
        Case Else:      ' An error other than 68, 75 or 76 has occurred.
                        ' Display the error number and the error text.
            MsgBox "Error # " & Err & " : " & Err.Description
    End Select
End Sub
```

При возникновении ошибки в процедуре, происходит одно из следующих действий:

- Если кодом ошибки является 68 или 75, выдается сообщение "**Ошибка при чтении диска G**" и макрос будет завершен.
- Если ошибка имеет код 76, то будет отображаться сообщение "**Указанный путь не найден**" и макрос будет завершен.
- Если ошибка имеет код, отличный от 68, 75 или 76, то выдается сообщение в формате: **«Ошибка < номер >: < текст ошибки >»** и макрос завершит работу.

Если книга **Book1.xls** успешно открыта, будет отображаться сообщение, показывающее целевую книгу, и макрос прекращает работу, так как имеется оператор **Exit Sub**, перед меткой обработчика ошибок **ErrHandler**.

Макрос, который использует оператор Resume

В данном примере используется оператор **Resume** для возобновления выполнения макроса на основе выбора пользователя, который он делает в результате прочтения сообщения об ошибке.

```
Sub MyMacro()
  Dim Result as Integer
  Dim ErrMsg as String
  Dim MyWorkbook as Workbook
  ' Run the Error handler "ErrHandler" when an error occurs.
  On Error GoTo Errhandler
```

```
  ChDrive "G:"
  ChDir "G:\"
  ChDir "G:\XLfiles"
  Workbooks.Open "Book1.xls"
NewWorkbook:
  ' Disable the error handler.
  On Error GoTo 0
  Set MyWorkbook = ActiveWorkbook
  MsgBox "The destination workbook is " & MyWorkbook.Name
  ' Exit the macro so that the error handler is not executed.
  Exit Sub
Errhandler:
  Select Case Err
    Case 68, 75:  ' Error 68: "Device not available"
                  ' Error 75: "Path/File access error"
      ErrMsg =  "There is an error reading drive G. Please " & _
                "insert a disk and then press OK to continue or " & _
                "press Cancel to end this operation."
      Result = MsgBox(ErrMsg, vbOKCancel)
      ' Resume at the line where the error occurred if the user' clicks OK;
      ' otherwise end the macro.
      If Result = vbOK Then Resume
    Case 76:      ' Error 76: "Path not found"
      ErrMsg = "The disk in drive G does not have an XLFiles " & _
      "directory. Please insert the correct disk."
      Result = MsgBox(ErrMsg, vbOKCancel)
      ' Resume at the line where the error occurred if the user
      ' clicks OK; otherwise end the macro.
      If Result = vbOK Then Resume
    Case Else:  ' A different error occurred.
      ErrMsg = "An error has occurred opening " & _
      "G:\XLFiles\Book1.xls. Use the active workbook as the destination?"

      Result = MsgBox(ErrMsg, vbYesNo)
      ' Resume at the label "NewWorkbook" if the user clicks Yes;
      ' otherwise end the macro.
      If Result = vbYes Then Resume NewWorkbook
  End Select
End Sub
```

Если книга **Book1.xls** успешно открыта, сообщение отобразит целевую книгу как **Book1.xls** и макрос будет завершен, так как имеется оператор **Exit Sub** перед меткой обработчика ошибок **ErrHandler**. При возникновении ошибки в макросе, обработчик будет выполнять одно из следующих действий:

- Если ошибкой является 68 или 75, отображается следующее сообщение:

Ошибка при чтении диска G. Вставьте диск и нажмите OK для продолжения или кнопку "Отмена" для завершения этой операции.

При нажатии кнопки ОК в диалоговом окне макрос возобновляется в строке, где произошла ошибка. Если пользователь нажимает кнопку "Отмена", макрос будет завершен.

- Если ошибкой является 76, то будет отображаться следующее сообщение:

  Диск в дисководе G не содержит каталог XLFiles. Вставьте правильный диск.

  При нажатии кнопки ОК в диалоговом окне макрос возобновляется в строке, где произошла ошибка. Если пользователь нажимает кнопку "Отмена", макрос будет завершен.

- Если ошибка не является ошибкой 68, 75 или 76, отображается следующее сообщение об ошибке:

  Произошла ошибка при открытии книги G:\XLFiles\Book1.xls. Использовать другую активную книгу в качестве целевой книги?

  Если пользователь нажимает кнопку "Да" в диалоговом окне, макрос возобновляется в строке, с меткой **NewWorkbook**. Устанавливается ссылка на другую активную в данное время книгу, которая принимается как целевая. Если пользователь нажмет кнопку "Нет", макрос будет завершен.

Централизация обработки ошибок

Чтобы уменьшить объем общего кода в приложении, используют централизацию обработки ошибок. Можно централизовать обработку ошибок, создавая одну или несколько процедур для обработки типичных ошибок.

В данном примере приводится процедура с именем **ErrorHandling**, которая будет отображать сообщение, соответствующее номеру ошибки (**ErrorValue**), который был передан и, где это возможно, разрешать пользователю выбрать кнопку, чтобы указать, какие действия должны быть выполнены. В зависимости от выбора, который делает пользователь, процедура **ErrorHandling** будет возвращать значение (**ReturnValue**) для действий в вызывающую процедуру. **ReturnValue** может быть **Err_Exit** (выход из макроса, в котором возникла ошибка), **Err_Resume** (выполнение строки в макросе, в которой возникла ошибка) или **Err_Resume_Next** (продолжение выполнения макроса со строки, следующей за строкой, в которой произошла ошибка).

```
Public Const Err_Exit = 0
Public Const Err_Resume = 1

Sub ErrorHandling(ErrorValue As Integer, ReturnValue As Integer)
  Dim Result as Integer
  Dim ErrMsg as String
  Dim Choices as Integer
  Select Case ErrorValue
    Case 68:    ' Device  not available.
      ErrMsg = "The device you are trying to access is either " & _
               "not online or does not exist. Retry?"
      Choices = vbOKCancel
    Case 75:    ' Path/File access error.
      ErrMsg = "There is an error accessing the path and/or " & _
               "file specified. Retry?"
      Choices = vbOKCancel
    Case 76:    ' Path not found.
      ErrMsg = "The path and/or file specified was not found. Retry?"
      Choices = vbOKCancel
    Case Else:  'An error other than 68, 75 or 76 has occurred
      ErrMsg = "An unrecognized error has occurred ( " & _
                Error(Err) & " ). The macro will end."
      MsgBox ErrMsg, vbOKOnly
      ReturnValue = Err_Exit
      Exit Sub
  End Select
  ' Display the error message.
  Result = MsgBox (ErrMsg, Choices)
  ' Determine the ReturnValue based on the user's choice from MsgBox.
  If Result = vbOK Then
    ReturnValue = Err_Resume
  Else
    ReturnValue = Err_Exit
  End If
End Sub
```

Следующий макрос показывает, как можно использовать процедуру **ErrorHandling** при возникновении ошибки:

```
Sub MyMacro()
  Dim Action As Integer
  ' Run the Error handler "ErrHandler" when an error occurs.
  On Error GoTo Errhandler
  ChDrive "G:"
  ChDir "G:\"
  ChDir "G:\XLFiles"
  Workbooks.Open "Book1.xls"
  ' Exit the macro so that the error handler is not executed.
  Exit Sub
```

```
Errhandler:
  ' Run the ErrorHandling macro to display the error and to
  ' return a value for Action which will determine the appropriate
  ' action to take (Resume the macro or end the macro)
  ErrorHandling (Err, Action)
  If Action = Err_Exit Then
    Exit Sub
  ElseIf Action = Err_Resume Then
    Resume
  Else
    Resume Next
  End If
End Sub
```

Таким образом, вместо распределенных по коду приложения (**MyMacro**) инструкций обработчика ошибок, получаем более упорядоченный – **структурированный** код, в котором все инструкции обработчика собраны в отдельной процедуре. В самом же макросе остаются только инструкция вызова обработчика ошибок и короткий код для анализа действий пользователя. Это улучшает восприятие исходного текста программы.

Обработка прерываний пользователя

Пользователь может прервать выполнение VBA-процедуры, нажав, к примеру, клавиши CTRL + BREAK. При этом нет гарантий **корректного** завершения работы приложения. Для предотвращения возможных отрицательных последствий подобных действий, можно отключать пользовательские прерывания для процедур готовых приложений. Например, если в качестве host-приложения используется MS Excel, то возможно полностью игнорировать прерывания пользователя, установив свойство MS Excel – **EnableCancelKey** в **xlDisabled**. В этом состоянии MS Excel игнорирует все попытки пользователя прервать выполнение процедуры.

Для того чтобы предотвратить состояние постоянного отключения прерываний пользователя – всякий раз, когда процедура завершает свое выполнение, MS Excel восстанавливает значение по умолчанию свойства **EnableCancelKey**, устанавливая его в **xlInterrupt**. Если это необходимо сделать раньше, чем процедура завершится, измените значение свойства **EnableCancelKey** на **xlInterrupt**.

Однако можно оставить пользователю постоянную возможность выполнять прерывание, перехватывая последнее в процедуре. Для этого свойство host-приложения – **EnableCancelKey**, необходимо установить в **xlErrorHandler**. Когда это свойство задано, прерывание пользователя будет представлено как сгенерированная ошибка времени выполнения с номером 18. Эту Run-time error можно перехватить с помощью оператора **On Error** и обработать событие (прерывание пользователя) соответствующим образом.

Можно обработать ошибку, чтобы остановить процедуру и корректно выйти из программы. Если в обработчике используется оператор **Resume** для продолжения процедуры после перехвата, прерывание игнорируется.

При обработке ошибки с номером 18, можно закрыть файлы, безошибочно отключиться от общих ресурсов или восстановить измененные переменные перед возвращением управления приложению и т.п.

Ниже приведена процедура, которая требует **большого** периода времени для завершения. Если пользователь прерывает процедуру, то прерывание пользователя перехватывается с уведомлением его о том, что выполнение процедуры может быть продолжено или фактически прекращено.

```
Sub ProcessData()
  'Установка перехвата прерывания пользователя, как обычного перехвата
   'ошибки времени выполнения (run-time error).
  On Error GoTo UserInterrupt
  Application.EnableCancelKey = xlErrorHandler
  'Выполнение какой-либо длительной работы.
  For x = 1 to 1000000
     For y = 1 to 10
       ' Операторы вычислений
     Next y
  Next x
  Exit Sub
UserInterrupt:
  If Err = 18 Then
    If MsgBox ("Остановить выполнение?", vbYesNo) = vbNo Then
      'Продолжение выполнения процедуры со строки, на которой она
      'была прервана.
      Resume
    Else ' При нажатии кнопки Yes в окне MsgBox.
      'Уведомление в том, что выполнение процедуры прервано пользователем.
      MsgBox "Error # " & Err & " : " & Err.Description
    End If
  End If
End Sub
```

После запуска макроса **ProcessData** и последующего одновременного нажатия клавиш CTRL + BREAK, появляется **окно сообщения**, предлагающее пользователю нужно ли остановить выполнение процедуры. Если нажать кнопку Yes, появится **другое** окно сообщения с уведомлением о прерывании выполнения процедуры пользователем. После нажатия кнопки OK в этом окне сообщения, макрос завершает работу. Если нажать кнопку NO в первом окне сообщения, макрос по-прежнему будет выполняться.

1.5. Блок Try-Catch-Finally

Блок кода **Try-Catch-Finally** – это способ написания **структурных** обработчиков ошибок в VB.NET и других языковых средах группы **.**NET. Хотя можно по-прежнему использовать код VB 6, включающий инструкции **On Error, Resume,** Resume Next и др.

При выполнении приложения время от времени происходят ошибки. **Исключение** (exception) это аварийное состояние, которое возникает в кодовой последовательности во время ее выполнения. Другими словами, исключение — это ошибка **времени выполнения** (run-time error). Оператор **Try-Catch-Finally** используется при обработке исключений. Он позволяет выполнить специфицированный блок кода при генерации указанного исключения. Оператор **Try** указывает на начало обработчика ошибок.

Вы помещаете оператор Try в процедуре непосредственно перед оператором, о котором вы беспокоитесь, а оператор Catch следует непосредственно за ним и содержит операторы, которые вы хотите выполнить, если произойдет ошибка времени выполнения.

Когда исключение генерируется внутри блока **Try**, то оно **перехватывается** соответствующим блоком **Catch** для последующей обработки. Если в блоке **Try** не произошла генерация исключений, то выполняется необязательный блок **Finally**. После этого управление передается оператору, следующему за инструкцией **End Try**, завершающей оператор **Try-Catch-Finally**.

Базовый (упрощенный) синтаксис блока кода **Try-Catch-Finally** можно представить в виде:

```
Try
  Операторы, которые могут вызвать ошибку времени выполнения
Catch
  Операторы, которые выполняются, если ошибка времени выполнения происходит
Finally
  Дополнительные операторы, выполняемые независимо от возникновения ошибки
End Try
```

Try, Catch и **End Try** – это обязательные ключевые слова, а **Finally** и операторы, которые стоят за ним, необязательны. Заметьте, что операторы, находящиеся между ключевыми словами **Try** и **Catch**, иногда называют защищенным кодом, так как любые ошибки времени выполнения, возникающие в этих операторах, не приведут к обрушению программы. (Вместо этого Visual Basic выполняет операторы обработки ошибок, расположенные в блоке кода **Catch**.)

Ссылка на обрабатываемое в блоке **Catch** исключение может дополняться необязательным ключевым словом **When**, специфицирующим условие, при котором происходит перехват исключения. После ключевого слова **when** должно располагаться любое логическое выражение, а сгенерированное исключение

перехватывается тогда, когда это логическое выражение возвращает значение True.

В операторе **Try-Catch-Finally** может быть любое число блоков **Catch**. Вот полный синтаксис блока обработки исключений:

```
Try
[try Statements]
Catch_1 [exception_1 [As type_1]] [When expression_1]
catchStatements_1
[Exit Try]
Catch_2 [exception_2 [As type_2]] [When expression_2]
catchStatements_2
[Exit Try]
...

Catch_n [exception_n [As type_n]] [When expression_n]
catchStatements_n
[Exit Try]
[Finally
[finallyStatements]]
End Try
```

Здесь:

● **tryStatements** — блок операторов, в котором может быть сгенерировано исключение.

● **Catch** — ключевое слово, идентифицирующее блок операторов **catchStatements,** выполняемых при генерации исключения **exception**, имеющего указанный тип **type**. При перехвате исключения для его более детальной спецификации можно использовать необязательное ключевое слово **When** с логическим выражением **expression**, конкретизирующим ситуацию.

● **Exit Try** — необязательное ключевое слово, которое подразумевает немедленный выход из оператора **Try-Catch-Finally** без выполнения блока **Finally**.

● **Finally** — необязательное ключевое слово, идентифицирующее блок операторов **finallyStatements,** который выполняется независимо от того, отработал ли хотя бы один блок **Catch**.

Следующий код дает пример перехвата исключений. Делается проверка исключения **overfiowException**, генерируемого при переполнении. Оно, в данной ситуации, может быть сгенерировано, когда значение переменной **i** равно **0**, т.е. при делении на ноль, или когда начальное значение переменной **y** слишком большое, как в приводимом ниже коде. Для того чтобы разграничить эти два сценария, в операторе **Catch,** в первом случае, имеется условие **When**, идентифицирующее ситуацию деления на ноль.

Кроме того, предусмотрена обработка непредвиденно возникших исключений путем размещения дополнительного оператора **Catch** без спецификации перехватываемого исключения.

```
Dim x As Integer = 5
Dim i As Integer = 1000000
Dim y As Integer = 1000000000
Try
   y = i * y
   x /= y
Catch ex As System.OverflowException  When y = 0
   Debug.WriteLine ("Деление на ноль")
Catch ex As System.OverflowException
   Debug.WriteLine ("Слишком большое число")
Catch
   Debug.WriteLine ("Неописанное исключение")
Finally
   Beep( ) ' Простейший звуковой сигнал
End Try
```

1.6. Примеры процедур, использующих обработку исключений

Ошибки путей и дисководов

В следующем примере продемонстрирована обычная ситуация возникновения ошибки времени исполнения – проблема с отсутствием доступа к требуемому источнику данных, - графическому файлу, содержащем изображение, открывающееся в объекте вывода изображений на windows-форме.

```
Try
    PictureBox1.Image = System.Drawing.Bitmap.FromFile     ("G:\Fileopen.bmp")
Catch
    MsgBox("Пожалуйста, вставьте диск в дисковод G!")
End Try
```

Этот код программы демонстрирует наиболее распространенный способ использования блока кода **Try-Catch**. Он помещает проблемный оператор **FromFile** в блок кода **Try**, и теперь при возникновении ошибки выполняются операторы, находящиеся в блоке кода **Catch**. Блок кода **Catch** просто отображает окно сообщения. Блок кода Try-Catch не содержит оператора **Finally**, так что обработчик ошибок завершается ключевыми словами **End Try**.

Использование блока Finally

Как отмечалось ранее при рассмотрении синтаксиса оператора **Try-Catch**, в этом операторе можно использовать дополнительный блок **Finally**. Этот блок исполняет операторы независимо от того, как компилятор выполняет блоки **Try** или **Catch**. Другими словами, независимо от того, приводят ли операторы в блоке **Try** к ошибке времени исполнения, есть необходимость в запуске кода, каждый раз по завершении действия обработчика ошибок. Например, можно обновить переменные или свойства, отобразить результаты вычислений в окне сообщения, выполнить очистку переменных или отключить ненужные объекты формы и т.п.

В следующем примере продемонстрировано, как работает блок **Finally**, отображающий второе окно сообщения, не зависимо от того, привел ли метод **FromFile** к ошибке времени выполнения, или нет.

```
Try
    PictureBox1.Image = System.Drawing.Bitmap.FromFile    ("G:\Fileopen.bmp")
Catch
    MsgBox("Пожалуйста, вставьте диск в дисковод G!")
Finally
    MsgBox("Обработка ошибок выполнена")
End Try
```

Оператор **Finally** указывает компилятору, что завершающий блок кода должен выполняться независимо от того, обработана ошибка времени исполнения, или нет. После оператора **Finally** вставлена функция **MsgBox**, которая отображает тестовое сообщение. Хотя это и полезно **для тестирования**, в законченном приложении блок кода **Finally** можно использовать для обновления значений переменных или свойств, отображения данных или выполнения других операций.

Проверка нескольких условий ошибок времени выполнения

С ростом сложности программного кода требуются более сложные обработчики ошибок **Try-Catch**, которые будут отслеживать различные ошибки времени выполнения и управлять необычными ситуациями обработки ошибок. **Try-Catch** поддерживает эту возможность с помощью:

- разрешения ввода нескольких строк кода в каждом из блоков **Try**, **Catch** или **Finally**;
- использования синтаксиса **Catch When**, который проверяет указанные условия возникновения ошибок;

● написания вложенных блоков кода **Try-Catch**, которые могут быть использованы для создания сложных, но надежных обработчиков ошибок.

В дополнение к этому специальный объект обработки ошибок – **Err** позволяет определять и обрабатывать в программах конкретные ошибки времени выполнения.

В следующем блоке кода обработчика ошибок **Try...Catch** первоначальный оператор **FromFile** – тот же, что и в коде, который использовался в предыдущих упражнениях, но операторы **Catch** изменились:

```
Try
    PictureBox1.Image = System.Drawing.Bitmap.FromFile     ("G:\Fileopen.bmp")
Catch When Err.Number = 53 'При ошибке File Not Found
    MsgBox("Проверьте правильность указания пути к файлу и наличие диска")
Catch When Err.Number = 7 'При ошибке Out Of Memory
    MsgBox("Это действительно точечный рисунок?", , Err.Description)
Catch
    MsgBox("Проблема при загрузке файла", , Err.Description)
End Try
```

Запись **Catch When** используется в этом обработчике ошибок дважды, и каждый раз она используется со свойством **Err.Number** и проверяет, сгенерировал ли блок **Try** конкретный номер ошибки времени выполнения, или нет.

Если свойство **Err.Number** содержит номер 53, это означает, что во время выполнения процедуры открытия файла произошла ошибка времени выполнения **"File Not Found"**, и в окне сообщения отображается сообщение "Проверьте правильность указания пути к файлу и наличие диска".

Если свойство **Err.Number** содержит номер 7, то произошла ошибка **"Out of Memory"** – вероятно, она стала результатом загрузки файла, который не является файлом изображения (например, если случайно, пытаются открыть в объекте вывода изображений с помощью метода **FromFile** документ Microsoft Word.)

Последний оператор Catch обрабатывает все остальные ошибки времени выполнения, которые потенциально могут возникнуть во время открытия файла – это общий блок "перехвата всего", который выводит в окне сообщения универсальное сообщение об ошибке, а в строке его заголовка выводит сообщение об ошибке, содержащееся в свойстве **Err.Description**.

Оператор **Catch When** обладает большой мощью. В сочетании со свойствами **Err.Number** и **Err.Description**, он позволяет писать сложные обработчики ошибок, которые распознают и реагируют на несколько типов ошибок.

Генерация собственных ошибок

Для тестирования блоков обработки ошибок выполнения можно искусственно сгенерировать исключения. Эта технология называется вбрасыванием или генерацией исключений. Для этого используется метод **Err.Raise** с одним из номеров ошибок. Например, следующая запись использует метод **Raise** для создания ошибки времени выполнения **Disk Full** (диск переполнен), а затем обрабатывает эту ошибку с помощью оператора **Catch When**:

```
Try
  Err.Raise    61 'генерируем ошибку Disk Full (диск переполнен)
Catch When Err.Number = 61
  MsgBox ("Ошибка: Диск переполнен")
End Try
```

Использование вложенных блоков Try-Catch

В обработчиках ошибок можно использовать вложенный блок кода **Try-Catch**. Например, если первая попытка прочитать данные с дисковода оказалась неудачной и сгенерировала ошибку времени выполнения, следующий обработчик ошибок использует для единственной повторной попытки выполнения операции второй блок **Try-Catch**:

```
Try
   PictureBox1.Image = System.Drawing.Bitmap.FromFile  ("G:\Fileopen.bmp")
Catch
   MsgBox ("Вставьте дискету в дисковод G, и нажмите OK!")
     Try
        PictureBox1.Image = System.Drawing.Bitmap.FromFile  ("G:\Fileopen.bmp")
     Catch
         MsgBox("Загрузка файла невозможна")
         Button1.Enabled = False
       End Try
End Try
```

Если пользователь после появления сообщения с запросом вставит диск в дисковод, второй блок Try откроет файл без ошибки. Однако если связанная с файлом ошибка времени выполнения появится снова, второй блок **Catch** выведет сообщение о том, что загрузка файла теперь невозможна, и отключит кнопку.

В основном, вложенные обработчики ошибок **Try-Catch** хорошо работают, когда не требуется совершать много повторных попыток. Если требуется выполнить проблемное действие много раз, используйте переменную для подсчета попыток, или разработайте функцию, которая будет содержать обработчик ошибок и которая может быть вызвана из процедур событий много раз.

Сравнение обработчиков ошибок с защитными технологиями

Обработчики ошибок являются не только механизмом для защиты программы от ошибок времени выполнения. Например, следующий код программы, чтобы проверить перед открытием файла его наличие на диске, использует метод **File.Exists** из пространства имен System.IO библиотеки классов .NET Framework:

```
If File.Exists  ("G:\Fileopen.bmp") Then
   PictureBox1.Image = _
   System.Drawing.Bitmap.FromFile ("G:\Fileopen.bmp")
Else
   MsgBox("Не могу найти Fileopen.bmp на диске G.")
End If
```

Примечание: В самое начало кода программы для формы необходимо включить оператор **Imports**, создающий ссылку на библиотеку классов .NET Framework для использования объектов, свойств и методов из этой библиотеки:

```
Imports System.IO
```

Оператор **If-Then** не является обработчиком ошибок, так как он **не предотвращает** остановку программы из-за ошибки времени выполнения. Это просто методика проверки, которую называют **защитным программированием**. Этот оператор использует удобный метод из библиотеки классов .NET Framework, чтобы проверить возможность выполнения файловой операции, прежде чем на самом деле пытаться ее выполнить. В данном конкретном случае, поверка наличия файла с помощью метода .NET Framework работает **быстрее**, чем ожидание, пока Visual Basic сгенерирует исключение и восстановится после ошибки времени выполнения с помощью обработчика ошибок.

Когда нужно использовать защитные методики программирования, а когда обработчики ошибок? Ответ зависит от того, насколько часто, по вашему мнению, будет возникать проблема с операторами, которые вы планируете использовать.

Если исключение или ошибка времени выполнения будет возникать довольно редко, скажем, менее чем в 25% случаев исполнения данного конкретного кода,

то использование обработчика ошибок может оказаться наиболее эффективным способом. Обработчики ошибок также более удобны, если у вас более одного проверяемого условия, и если вы хотите предоставить пользователю различные возможности реакции на ошибку.

Однако если существует реальная возможность того, что данный кусок кода будет приводить к ошибкам времени исполнения в более чем 25% случаев его исполнения, то обычно защитная методика программирования является наиболее эффективным способом управления.

Как уже было сказано при обсуждении блока кода **If-Then**, метод **File.Exists** на самом деле работает быстрее, чем использование обработчика ошибок **Try-Catch**, так что имеет смысл использовать защитную методику программирования, если большое значение имеет **производительность**. Наконец, лучше всего использовать в коде комбинацию защитной методики программирования и структурной обработки ошибок.

2. Отбор данных по критерию в приложениях баз данных

Запрос – это команда, осуществляющая выборку записей с данными из одного или нескольких полей, принадлежащих одной или нескольким таблицам базы данных. Отбираемые данные можно подвергать действию одного или нескольких условий, называемых ***критериями***, служащими для ограничения общего объема отбираемых данных.

2.1. Использование SQL-запросов для доступа к данным

Запросы в VBA обычно строятся на основе ***структурированного языка запросов SQL****(Structured Query Language)*. Это стандартный язык для выборки информации и осуществления других операций над базами данных. Команды языка SQL делятся на две большие категории: команды манипулирования данными (DML) и команды определения данных (DDL).

SQL-команды позволяют отображать для пользователя не всю таблицу, а какую-то выборку данных из этой таблицы, или даже из нескольких таблиц. Выборка может производиться по довольно сложным критериям, с одновременным подведением итогов по выбираемым данным.

Для рассмотрения приводимых ниже примеров базовых синтаксических конструкций SQL-команд, будем использовать таблицу базы данных **Phone**:

Phone

ID	LastName	FirstName	Phone	City	Age
1	Иванов	Иван	495 123 32 24	Москва	35
2	Петров	Петр	865 342 43 34	Ставрополь	24
3	Сидоров	Сидор	812 324 45 24	С-Петербург	55
4	Антонов	Антон	423 355 76 29	Владивосток	40
5	Борисов	Борис	421 591 55 67	Хабаровск	31
6	Галкина	Галина	383 765 35 34	Новосибирск	56
7	Григорьев	Григорий	499 879 22 11	Москва	29
8	Васильева	Вероника	861 672 76 33	Новороссийск	33

Оператор **SELECT**

Оператор **SELECT** позволяет произвести выборку записей по указанному критерию. Этот оператор возвращает построенную выборку данных из одной или нескольких таблиц базы данных.

Возвращение всех записей таблицы

Если требуется вернуть все записи таблицы (в данном случае **Phone)**, то можно воспользоваться командой:

```
SELECT * FROM Phone
```

или, что равносильно

```
SELECT ALL * FROM Phone
```

Возвращение всех записей одного поля

Для того чтобы вернуть набор всех записей одного поля таблицы, например **LastName,** примените следующую команду:

```
SELECT LastName FROM Phone
```

Возвращение всех записей двух полей

Если требуется вернуть все записи двух полей таблицы, например **LastName** и **Phone,** воспользуйтесь командой:

```
SELECT LastName, Phone FROM Phone
```

Упорядочивание элементов: предложение **ORDER BY**

Добавление в оператор **SELECT** предложения **ORDER BY** обеспечивает упорядочивание набора выбранных записей. Добавление ключевого слова **ASC** или

DESC после имени поля задает порядок сортировки по этому полю (возрастающий или убывающий)

Упорядочивание записей по одному полю

Следующий оператор возвращает всю таблицу **Phone,** записи которой упорядочены по полю **LastName** в порядке возрастания:

```
SELECT * FROM Phone ORDER BY LastName ASC
```

Упорядочивание записей по двум полям

Оператор, приводимый ниже, возвращает записи двух полей (**LastName** и **Age)** таблицы **Phone,** причем записи упорядочены по полю **LastName** в порядке возрастания, а затем по полю **Age** в порядке убывания:

```
SELECT LastName, Age FROM Phone ORDER BY LastName ASC, Age DESC
```

Выборка записей по критерию: предложение **WHERE**

Добавление в SQL-команду предложения **WHERE** обеспечивает проведение выборки по критерию. В предложении **WHERE** допускается использование ***простых*** операций сравнения (**=, <>, <, >, <=, >=**), например, **LastName = 'Иванов'** или **Age > 20**, либо ***составных*** с использованием логических операторов **AND** и **OR**.

Ссылка на строки и даты в операциях сравнения

Обратите внимание на то, что все строки в операциях сравнения заключаются в простые кавычки, например, **LastName = 'Иванов'**. Подобным же образом даты должны обрамляться символом "#", например, 'Дата рождения' > #**15.06.2016**#.

Предложение WHERE и выборка записей по простому критерию

Первая из следующих двух команд возвращает все записи с московскими телефонами, а вторая – всех адресатов, которые не младше 20 лет:

```
SELECT * FROM Phone WHERE City = 'Москва'
SELECT * FROM Phone WHERE Age >= 20
```

Выборка записей по составному критерию

В предложении **WHERE** допустимо применение составных операций с использованием логических операторов **AND** и **OR.** Например, следующая команда возвращает как все записи с московскими телефонами, так и те, у которых возраст абонентов лежит в диапазоне от 21 года до 30 лет:

```
SELECT * FROM Phone
   WHERE (City  =  'Москва')  OR  (Age >= 20  AND  Age <=  30)
```

Выборка записей, значения специфицированного поля которых принадлежат указанному диапазону или лежат за его пределами

В конструкции **WHERE** допустимо использование операций сравнения **BETWEEN AND или NOT BETWEEN AND** для выборки записей, значения специфицированного поля которых принадлежат указанному диапазону или лежат за его пределами. В первом из следующих двух примеров возвращаются все записи с возрастом абонентов от 21 года до 30 лет, а во втором – вне этого диапазона:

```
SELECT * FROM Phone WHERE  (Age BETWEEN  21  AND  30)
SELECT * FROM Phone WHERE  (Age NOT BETWEEN  21  AND  30)
```

Выборка записей, значения специфицированного поля которых принадлежат указанному множеству

В конструкции **WHERE** допустимо использование операции сравнения **IN** для выборки записей, значения специфицированного поля которых принадлежат указанному множеству. В следующем примере возвращаются все записи с московскими и петербургскими номерами телефонов:

```
SELECT * FROM Phone WHERE  (City  IN  ('Москва', 'С-Петербург'))
```

Выборка записей по шаблону

В конструкции **WHERE** допустимо использование операции сравнения **LIKE** для выборки записей, значения специфицированного поля которых имеют определенный шаблон. Создавая шаблоны, можно использовать символ " **%** " для указания множества символов. Операция **NOT LIKE** задает все записи, не подходящие к указанному шаблону. Например, первая из следующих двух команд воз-

вращает все записи с фамилиями, оканчивающимися на "ва" – записями с женскими фамилиями, а вторая команда – все остальные:

```
SELECT * FROM Phone WHERE  LastName  LIKE  ' %ва '
SELECT * FROM Phone WHERE  LastName  NOT LIKE  ' %ва '
```

Выбор различных записей

Использование предиката **DISTINCT** в операторе **SELECT** позволяет отображать только различные записи. Например, следующая команда выводит всех абонентов с различными фамилиями:

```
SELECT  DISTINCT  LastName  FROM  Phone
```

Нахождение общего числа записей, возвращаемого запросом

Для определения количества записей, удовлетворяющих специфицированному критерию, достаточно воспользоваться функцией **COUNT** со значением " * " в качестве ее параметра. Например, следующая команда возвращает число московских абонентов из таблицы:

```
SELECT  COUNT( * )  FROM  Phone  WHERE  City = 'Москва'
```

Статистические функции

Оператор **SELECT** можно применять для выполнения вычислений со значениями, которые хранятся в таблицах. При этом используются статистические функции (приведены в таблице ниже), которые вычисляют соответствующие значения поля в записях, отвечающих условию **WHERE.**

Итоговые функции языка SQL

Функция	Возвращаемое значение
SUM ([DISTINCT] *exprs*)	**Сумма (неповторяющихся) значений**
AVG ([DISTINCT] *exprs*)	**Среднее значение (неповторяющихся) величин**
COUNT ([DISTINCT] *exprs*)	**Число (неповторяющихся) значений, не равных NULL**
COUNT (*)	**Число выбранных строк**
MIN (*exprs*)	**Минимальное значение в данном выражении**
MAX (*exprs*)	**Максимальное значение в данном выражении**

Следующая команда возвращает число московских абонентов, а также их минимальный, максимальный и средний возраст:

```
SELECT COUNT ( Age ), MIN ( Age ), MAX ( Age ), AVG ( Age )
   FROM  Phone  WHERE  City = 'Москва'
```

Определение состояния ячеек поля таблицы

Значение данных в ячейке поля таблицы отражает не только отсутствие или наличие информации в ней, но и указывает на то, что некоторая информация неизвестна. В SQL для указания на такое неопределенное состояние используется значение NULL. В каком-то смысле это значение переводит SQL в язык трехзначной логики: возвращаемым значением может быть как истина или ложь, так и неопределенная величина – **NULL**.
Определить, содержит ли поле значение NULL, можно с помощью операции сравнения **IS**, а то, что там такового нет – с помощью операции **IS NOT**. Например, следующая команда создает набор всех записей с непустым полем **Age**:

```
SELECT  LastName,  Age  FROM  Phone  WHERE  Age  IS NOT  NULL
```

Оператор **DELETE**

Оператор **DELETE** применяется для создания функционального запроса на удаление из таблицы определенных записей. Функциональный запрос не возвращает группу записей в виде динамического набора, как это делает оператор **SELECT.** Оператор **DELETE** имеет следующий синтаксис:

DELETE FROM ***имяТаблицы*** [**WHERE** ***условие*** **]**

Действие оператора **DELETE** нельзя отменить. Например, следующий код удаляет из таблицы **Phone** запись, у которой поле **LastName** имеет значение **Иванов**

```
DELETE  FROM Phone WHERE LastName = 'Иванов'
```

Оператор **INSERT**

Подобно **DELETE** оператор **INSERT** создает функциональный запрос. В сочетании с оператором **SELECT** он используется для добавления в таблицу новых записей. Оператор **INSERT** имеет следующий синтаксис:

INSERT INTO ***имяТаблицы*** **[(*списокСтолбцов*)] VALUES (*списокДанных*)**

Например, следующая команда вставляет в поля **LastName** и **Phone** таблицы **Phone** значения **Иванова и 499 485 34 77:**

INSERT INTO Phone (LastName, Phone) VALUES ('Иванова', '499 485 34 77')

Оператор **UPDATE**

Еще одной разновидностью функционального запроса является оператор **UPDATE.** С его помощью можно изменять значения указанных полей таблицы. Оператор **UPDATE** имеет следующий синтаксис:

UPDATE *имяТаблицы* SET [*поле* = *значение* [, *поле* = *значение*] ...] [WHERE *условие*]

Например, следующая команда обновляет запись со значением поля **LastName,** равным **Иванова:**

```
UPDATE Phone SET Phone = '495 485 34 77', Age = 31
  WHERE LastName = 'Иванова'
```

Создание групп записей

Возможность создания групп записей позволяет построить набор записей, в котором для каждого значения указанного поля будет только одна запись. Например, если сгруппировать таблицу **Phone** по абонентам, то для каждого города получим одну выходную запись. Если выполнить группирование по городам, то для каждой страны получим одну выходную запись и т.д.

Такая организация записей особенно удобна в сочетании с описанными выше статистическими функциями. В этом случае упрощается получение итоговой информации, к примеру, для городов, определенного товара, торгового агента и т.п.

Следующий запрос возвращает суммарный возраст абонентов в каждом городе:

SELECT City, SUM (Age) AS [Sum- Age] FROM Phone GROUP BY City

В этой инструкции используется еще одно ключевое слово – **AS.**

Создание псевдонима с использованием предложения **AS**

В SQL имеется возможность переименования полей в запросе посредством определения их псевдонимов (alias) при помощи предложения **AS.** Псевдонимы используются по двум причинам:
- если необходимо упростить громоздкие имена полей;
- если запрос создает столбец (как это и произошло в предыдущем примере), который заполняется в результате некоторых вычислений, и новому столбцу необходимо задать имя.

Предложение **HAVING**

Предложение **HAVING** позволяет создать результирующие наборы, содержащие список промежуточных итогов. В следующем примере запрос возвращает суммарный возраст абонентов в каждом городе, при условии наличия в таблице более одного абонента:

```
SELECT City, SUM (Age) AS [Sum- Age] FROM Phone GROUP BY City
   HAVING COUNT (Age) > 1
```

2.2. Примеры обращения к данным с помощью SQL

Поиск требуемых записей

Самый простой способ найти требуемую запись в результирующем наборе заключается в использовании методов группы Find. Данные методы находят в объектах Recordset типа **динамических** и **статических** наборов записей первую, последнюю, следующую или предыдущую запись, удовлетворяющую заданным условиям, и делают найденную запись **текущей**.
Методы FindFirst, FindLast, FindNext и FindPrevious имеют следующий синтаксис:

recordset. {FindFirst | FindLast | FindNext | FindPrevious} ***условия***

Здесь:

recordset – объектная переменная, ссылающаяся на существующий объект Recordset. Аргумент *условия* – **строка**, используемая для поиска записи. Данный параметр аналогичен предложению **WHERE** в SQL-инструкции, но ключевое слово **WHERE** не используется в данном случае.

Если запись, отвечающая заданным условиям, не обнаружена, то состояние указателя текущей записи становится **неопределенным**, и свойству **NoMatch**

присваивается значение True. Всегда необходимо следить с помощью указанного свойства за тем, была ли успешной операция Find. Если запись обнаружена, свойство **NoMatch** принимает значение **False**. в противном случае необходимо явно установить указатель текущей записи на допустимую запись.

```
Sub find_first( )
   Dim searchfor As String
   Dim original_loc As String
   'В свойстве Bookmark хранится позиция указателя текущей записи.
   'Фиксируем первоначальное расположение указателя.
   original_loc = rs.Bookmark
   searchfor = InputBox ("Введите требуемый товар: ", "Выбор товара")
   On Error Resume Next
   rs.FindFirst " [НазваниеТовара] LIKE ' " & searchfor & " * ' "
   'Проверка состояния указателя текущей записи. Вывод данных в форме
   IF rs.NoMatch = True Then
      rs.Bookmark = original_loc
   End IF
   PopulateDialog
End Sub
'Связь полей результирующего набора с элементами управления в форме
'dlg – объектная переменная для формы, объявленная ранее
Sub PopulateDialog ( )
   dlg.TextBoxes ("txtProdName").text = rs.Fields ("НазваниеТовара").value
   dlg.TextBoxes ("txtProdPrice").text = rs.Fields ("Цена").value
End Sub
```

Отбор записей в результирующий набор

Наиболее **гибкий** способ состоит в отборе записей для результирующего набора, используя SQL-запрос. В этом случае можно не применять метод Seek, который работает только с табличными объектами Recordset, а также не использовать методы группы Find, имеющие ограниченное быстродействие.
Для поиска требуемой записи следует открыть результирующий набор. Затем проверить значение свойства **RecordCount** объекта **Recordset**, чтобы удостовериться, что записи, соответствующие критериям отбора, найдены.

```
Sub cmdFind_Click( )
   Dim curr_location As String
   Dim searchfor As String
   'Фиксируем первоначальное расположение указателя
   curr_location = rs.Bookmark
   searchfor = InputBox ("Введите требуемый товар: ", "Выбор товара")
   On Error Resume Next
   'Переопределяем ссылку на новый объект RecordSet, формируемый на основе SQL-запроса
   'из созданного ранее результирующего множества, основанного на запросе "Товары"
   Set rs = db.OpenRecordset ("SELECT * FROM [Товары] WHERE _
   [НазваниеТовара] LIKE ' " & searchfor & " * ' ")
```

```
    'Вывод данных в форме, если найдены требуемые данные
    IF rs.RecordCount > 0 Then
       PopulateDialog
    Else
       MsgBox "Невозможно найти указанный товар"
    'Возвращаем ссылку на первоначальный результирующий набор RecordSet
    Set rs = db.OpenRecordset ("Товары", dbOpenDynaset)
    'Вернем первоначальный результирующий набор в исходное состояние
    rs.Bookmark = curr_location
    End IF
End Sub

Sub PopulateDialog ( )
    dlg.TextBoxes ("txtProdName").text = rs.Fields ("НазваниеТовара").value
    dlg.TextBoxes ("txtProdPrice").text = rs.Fields ("Цена").value
End Sub
```

Источники информации

Содержание данного учебно-методического пособия (текстовые фрагменты, учебные примеры кодов) подготовлено на основе материалов, содержащихся в следующих литературных источниках:

1. Кузьменко В.Г. VBA. Эффективное использование – М.: Бином, 2012

2. Кузьменко В.Г. Базы данных в Visual Basic и VBA. Самоучитель. Серия: Для программистов и разработчиков – М.: Бином, 2004

3. Гарнаев А.Ю. Самоучитель Visual Studio .NET 2003 – СПб.: БХВ-Петербург, 2003

4. Гарнаев А.Ю. Visual Basic .NET: разработка приложений – СПб.: БХВ-Петербург, 2002

5. Хальворсон М. Microsoft Visual Basic .NET 2003 – М.: Эком, 2004

6. https://www.visualstudio.com/

7. https://msdn.microsoft.com/en-us/library/5hsw66as.aspx