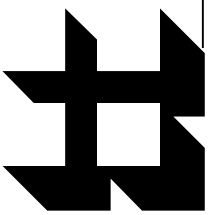

ГЛАВА 1

Введение в Visual FoxPro

Итак, вы решили написать программу на языке Visual FoxPro, но не знаете, как осуществить свое желание. С помощью этой книги вы сможете это сделать. Вместе с авторами вы пройдете путь от простого — к сложному, от небольших фрагментов кода к созданию собственного приложения. Цель этой книги — познакомить вас с удивительным миром Visual FoxPro.

Visual FoxPro — это объектно-ориентированный, визуально программируемый язык управления реляционными базами данных. Язык Visual FoxPro может применяться пользователями различных уровней подготовки: начинающий программист может создавать с помощью Visual FoxPro простые приложения в достаточно короткие сроки, опытный пользователь может воспользоваться широчайшими возможностями обработки информации с помощью полнофункциональной интегрированной среды программирования Visual FoxPro — объектно-ориентированным программированием (ООП), классами, COM-объектами, а также предоставляемым Visual FoxPro доступом к наиболее популярным серверам баз данных — Microsoft SQL Server, Oracle, DB2, Informix и т. д. Основоположником реляционной модели баз данных называют сотрудника компании IBM Е. Ф. Кодда, опубликовавшего в 1970 году статью о реляционной модели баз данных.

Что такое "реляционные базы данных" и почему они уже на протяжении многих лет пользуются высокой популярностью? В реляционных базах данных вся информация хранится в таблицах, строки таблицы называются записями, а столбцы — полями. Можно представить себе таблицу в виде телефонной или адресной книги, списка магазинов или пациентов больницы. Реляционные базы данных должны удовлетворять следующим требованиям:

- данные организованы в одну или несколько таблиц с уникальными строками. Уникальность строк обеспечивается первичным ключом (Primary key). Первичный ключ — это сочетание нескольких столбцов или один столбец, не содержащий повторяющихся значений;

- одна ячейка может содержать только одно значение. Это значит, что не может быть столбца "Телефоны", может быть столбец "Телефон". Если необходимо хранить много телефонов, то создается отдельная таблица;
- СУБД включает в себя поддержку набора таблиц — базы данных;
- кроме того, СУБД должна поддерживать виртуальные таблицы как альтернативный способ получения данных;
- реляционная СУБД позволяет работать с данными одновременно нескольким пользователям, включая в себя методы блокировки, благодаря которым операции над данными одного пользователя не отменяют результатов работы другого;
- реляционная СУБД должна позволять различать неопределенные (пропущенные) данные.

Язык Visual FoxPro невероятно богат своими возможностями. От версии к версии разработчики добавляли в него новые команды и функции, старые же команды при этом продолжали действовать, чтобы обеспечить совместимость с предыдущими версиями, что позволяло и позволяет переносить ранее созданные приложения в новую среду.

Элементы языка

К элементам языка Visual FoxPro относятся:

- команды;
- литералы и константы;
- переменные и массивы;
- процедуры и функции;
- операторы и выражения;
- условные операторы;
- циклы;
- директивы препроцессора;
- зарезервированные слова и системные переменные.

Команды

Команда — это инструкция компьютеру выполнить определенное действие. Команда может быть введена с клавиатуры, выбрана в меню или записана в код программы. Максимальная длина командной строки в Visual FoxPro 9.0 составляет 8192 байта, для переноса части команды на другую строку используется символ ; (точка с запятой). Каждая команда имеет название (имя). Некоторые команды состоят из одного имени, а некоторые имеют параметры, которые могут быть обязательными или необязательными. При описании синтаксиса команд необязательные параметры указы-

ваются в квадратных скобках. Поскольку Visual FoxPro позиционируется как язык управления реляционными базами данных, то в своем арсенале он имеет большое количество команд управления таблицами, базами данных, индексами, меню, окнами, отчетами и т. д.

В общем случае команды Visual FoxPro можно разделить на следующие категории:

- команды управления;
- команды управления данными;
- команды управления индексами;
- команды управления записями;
- команды установки отношений между таблицами;
- команды управления курсорами, таблицами и базами данных;
- команды управления средой окружения;
- команды управления файлами;
- команды управления меню;
- команды управления печатью и отчетами;
- команды для работы с клавиатурой и мышью;
- команды управления окнами;
- команды для работы в сети;
- команды управления массивами;
- команды для обработки ошибок;
- команды управления событиями;
- команды для работы с объектами;
- команды ввода-вывода;
- SET-команды;
- системные переменные.

Количество команд в Visual FoxPro — около полутысячи. Набор команд невероятно богат, кроме того, от версии к версии разработчики расширяют возможности языка.

Команды Visual FoxPro 9.0 сведены в таблицу и приведены в *приложении 2*. Также для ознакомления с командами или для поиска нужной команды можно использовать справочную систему, размещенную на компакт-диске: CHAR1\spravka\spravka.exe.

Литералы и константы

Литералы и константы — это самые простые составляющие любой программы, обычно они используются в виде "как есть". Литералом называют значение в явном представлении, например, число, строковое значение или дату (соответственно, это будет либо числовой литерал, либо строковой литерал, либо литерал даты). Литерал,

представляющий дату, берется в фигурные скобки. Для написания символьных литералов используются три вида символов: одинарные кавычки, двойные кавычки и квадратные скобки.

Например:

'*текст1*' "*текст2*" *[текст3]*

Примеры литералов приведены в табл. 1.1.

Таблица 1.1. *Примеры литералов*

12.08	Число фиксированное, не изменяется
{20.10.2005}	Фиксированная дата
array[2,2]	Массив фиксированной размерности
Messagebox('Ждите, идет расчет')	Это сообщение всегда одно и то же
.F.	Логический литерал

Константа — это именованный литерал. Если литералу присвоить имя, то все ссылки на это имя будут заменяться его значением. Это значение не может меняться в процессе выполнения программы. Значение константе может быть присвоено не только в процессе выполнения приложения, а и в период компиляции. Именованные константы объявляются в директиве препроцессора `#DEFINE`.

Для присвоения значения константе создайте текстовый файл с расширением h, например, `Def_Const.h`, внутри этого файла запишите директиву

`#DEFINE <имя константы> <ее значение>`

В вашу стартовую программу вставьте директиву `#INCLUDE Def_const.h`.

Символьные константы иногда называют строками.

Константы в Visual FoxPro могут быть также двоичными и шестнадцатеричными. Двоичный литерал начинается обязательно с *0h*, например *0h6A77RE*, и имеет неопределенную длину. Шестнадцатеричный литерал может иметь длину до 255 байтов и занимает 1 байт на каждый 16-ричный символ.

Переменные и массивы

Переменная представляет собой зарезервированное место в памяти компьютера для хранения значения. Переменные обозначаются именами — словами, используемыми для ссылки на значение, которое содержит переменная. Переменные могут изменять свои значения в ходе выполнения программы. Если переменной `Var1` присвоить значение `Var1=123`, то она будет числовой переменной. Если на каком-либо этапе программы изменить значение `Var1="Hello, World!"`, то она станет строковой переменной. Visual FoxPro относится к группе слаботипизированных языков, так называемых "weak-typing". Это значит, что одна и та же переменная может хранить разные типы

данных. Значение переменной можно присвоить оператором присваивания, он имеет такой синтаксис:

```
<Имя переменной> | <Имя массива> = <Выражение>.
```

Например: `My_var=10` или `My_var="Статистика"`

Кроме оператора присваивания, можно использовать команду `STORE`, которая используется для сохранения значения в переменной или элементе массива.

Команда имеет следующий синтаксис:

```
STORE <выражение> TO <список переменных> | <Список массивов>
```

При создании переменная всегда имеет тип `logical` со значением `False` и тип ее фиксируется (определяется) при первом присваивании переменной какого-либо значения. То есть, при первом присваивании переменная меняет свой тип на тип присваиваемых ей данных.

Типы данных

В Visual FoxPro 9.0 существуют следующие типы данных (табл. 1.2).

Таблица 1.2. *Типы данных*

Тип данных	Описание
`Blob`	Двоичные данные неопределенной длины. Начинается обязательно с 0h, например 0hABCD. Размер 4 байта в таблице. Хранится в memo-поле (FPT-файл). Не имеет установленной кодовой страницы. Размер ограничивается либо размером памяти, либо действует ограничение на размер таблицы в 2 Гбайт
`Character`	Символьный тип. Может содержать буквы, цифры, знаки пунктуации, пробелы, спецсимволы, нулевые байты. Максимальный размер символьной строки составляет 16 777 184 символа
`Currency`	Денежный тип, представляет собой 8-байтное значение валюты, для него перед числом устанавливается знак $ Максимальный размер составляет от — $922337203685477.5807 до $922337203685477.5807
`Date`	Хронологический тип, дата, содержит дни, месяцы, годы. Размерность 8 байт, может содержать даты от {^0001-01-01} до {^9999-12-31)
`DateTime`	Данные хронологического типа, содержит дату (дни, месяцы, годы) и время (часы, минуты, секунды). Размерность 8 байт. Диапазон значений может быть от {^0001-01-01} до {^9999-12-31) для даты и от 00:00:00 до 11:59:59 для времени (для сокращенного формата)
`Logical`	Логический тип, может принимать только два значения — "истина" (`.T.`) или "ложь" (`.F.`)
`Numeric`	Числовой тип, может содержать только целые либо десятичные числа. Размерность — 8 байт в памяти и от 1 до 20 байт в таблице. Диапазон значений от -.9999999999E+19 до +.9999999999E+20

Varbinary	Это двоичные величины или литералы фиксированной длины, не заполненные дополнительными нулевыми (0) байтами. Двоичные литералы начинаются с префикса 0h, сопровожденного строкой шестнадцатеричных чисел и не сопровождается кавычками в отличие от символьных строк
Variant	Данные типа Variant могут иметь любой из типов данных Visual FoxPro и нулевое значение. Как только значение будет загружено в Variant, Variant принимает тип данных, который он содержит. Данные типа Variant определены префиксом "e" в языковом синтаксисе

Довольно часто в программах приходится встречать переменные с префиксом m. перед именем переменной. Это означает, что это имя переменной памяти. В случае отсутствия каких-либо префиксов FoxPro предполагает, что речь идет о поле таблицы в текущей рабочей области. И только если в текущей рабочей области нет поля с таким именем, выполняется поиск среди переменных памяти.

Статус переменных (соглашение о написании имен переменных)

Для имен переменных существует несколько правил, которые необходимо соблюдать (свод этих правил носит название "Венгерская нотация"):

- имя переменной может содержать не более 254 символов;
- в имени переменной могут быть буквы и цифры, но начинаться она должна всегда с буквы;
- в имени переменной не могут присутствовать пробелы;
- в Visual FoxPro существуют имена, специально зарезервированные для системных переменных FoxPro. Как правило, они начинаются с символа "_";
- максимальное количество переменных в программе — 65 000;
- количество переменных по умолчанию — 1024;
- имя переменной не должно быть зарезервированным словом языка FoxPro или совпадать с его первыми четырьмя буквами.

Старайтесь давать переменным максимально информативные имена. Например, LdCurrentDat — пример максимально информативного имени переменной.

Для написания символьных констант используются 3 вида символов: одинарные кавычки, двойные кавычки и квадратные скобки. Например:

```
'текст1'   "текст2"  [текст3]
```

В строке

```
pcstring = 'текст1'+"текст2"+[текст3]
```

pcstring будет хранить данные символьного типа.

Если же в символьной константе или переменной нужно использовать кавычки не как ограничительные символы, а как часть текста, используйте кавычки другого типа, например " ' ".

Рассмотрим отличие символьной переменной от символьной константы.

Символьная переменная может хранить до 16 777 184 символов, а символьная константа ограничена 255 символами. Поэтому, чтобы создать символьную строку из набора символьных констант, нужно это учитывать, разбивая строку на фрагменты не длиннее 255 символов.

Следует быть внимательными при заполнении символьных полей в таблицах или курсорах. Если поле таблицы объявлено длиной 5 символов, а вы попытаетесь записать туда 6, шестой символ будет просто отброшен без всяких сообщений.

Статус переменных (область видимости)

Область видимости — это понятие об "области действия" переменной. Переменные должны быть правильно объявлены, в противном случае это может привести к ошибке в программе либо к неоднозначности результата. Переменные могут быть объявлены следующим образом:

- `PRIVATE <список переменных> ALL [LIKE <маска>] либо [EXCEPT <маска>]` — действуют в той программе, в которой они были объявлены, и в любых процедурах, вызываемых из данной. По окончании выполнения приватные (или закрытые) переменные освобождаются. Переменные, которые не были объявлены явно, по умолчанию считаются приватными.
- `LOCAL <список переменных>` — объявляет переменные, которые действуют только в той программе, в которой они объявлены, и не действуют даже в процедурах и функциях, вызываемых из этой программы. Такие переменные называют локальными.
- `REGIONAL <список переменных>` — объявляет региональные переменные и следует сразу за директивой `#REGION <номер региона от 0 до 31>`, которая определяет часть программы — регион. В этом регионе и действуют такие переменные.
- `PUBLIC <список переменных>` — глобальные, общедоступные переменные. Действуют в пределах всей программы, не уничтожаются по ее завершении. Такие переменные нужно освобождать по мере необходимости. Для освобождения используйте команду

  ```
  RELEASE <список переменных> ALL [/LIKE <маска> / EXCEPT <маска>]
  ```

Чтобы освободить все переменные, можно использовать команду `CLEAR ALL`.

Для именования переменных FoxPro не имеет значения регистр: верхний или нижний. Переменные `Myvar` и `myvar` будут считаться одной и той же переменной.

В связи с тем, что переменные можно не объявлять и по ходу выполнения функции записывать в одну и ту же переменную различные типы данных, в VFP принято явно указывать как тип переменной, так и область ее действия. Возьмите себе за правило верно именовать переменные прямо в начале процедуры, программы, функции и следить за их использованием, и вы избавитесь от ряда проблем при разработке про-

грамм. Существуют типичные правила именования переменных. В табл. 1.3 указаны префиксы переменных в зависимости от области видимости.

Таблица 1.3. *Префиксы наименований переменных*

Префикс в наименовании переменной	Значение
L	LOCAL (локальная)
G	PUBLIC (глобальная)
P	PRIVATE (закрытая)
T	PARAMETR (параметр)

В табл. 1.4 указаны префиксы переменных в зависимости от хранимого в ней типа данных, чтобы было легче ориентироваться в том, что же в ней хранится.

Таблица 1.4. *Второй префикс наименований переменных*

Префикс в названии переменной	Значение
A	Array (Массив)
C	Character (Символьный тип)
Y	Currency (Денежный тип)
D	Data (Дата)
T	DateTime (Дата и время)
B	Double (Число с плавающей точкой двойной точности)
F	Float (Вещественный тип)
L	Logical (Логический тип)
N	Numeric (Число)
O	Object (Объект)
U	Неопределенный тип

Если совместить префиксы из этих двух таблиц при именовании переменной, то легко будет разобраться, для чего она необходима.

Примеры:

- `lnCount` — локальная числовая переменная, по смыслу названия — счетчик;
- `glTrue` — глобальная логическая переменная;
- `pdDatR` — приватная переменная, содержит дату, предположительно — дату рождения или дату регистрации;

- `gcAdress` — глобальная символьная переменная, содержащая адрес.

Для определения типа переменной в программе существует такая замечательная функция, как `VARTYPE()`. Если предложить ей в качестве параметра имя переменной, то в результате ее выполнения получим символ, описывающий тип данных параметра. Синтаксис команды:

```
VARTYPE (выражение | флаг),
```

где флаг — флаг анализа нулевого значения.

Функция `TYPE <выражение>` выполняет то же самое, но аргумент ее должен быть заключен в кавычки.

Таблица 1.5. *Типы переменных, возвращаемые функцией* `VARTYPE()`

Значение	Тип данных
C	Символьный или memo
D	Дата
G	General
L	Логический
N	Числовой
O	Объект
Y	Currency
T	Дата-время
U	Неопределенный тип
X	NULL
Q	Blob

Массивы

Массив — это группа переменных памяти, с которыми обращаются как с единым целым. Массив, как и переменная, используется для хранения данных, и может быть представлен как электронная таблица, имеющая строки и столбцы. На пересечении строки и столбца находится ячейка, которая характеризуется индексом массива, т. е. номером строки и столбца, на пересечении которых она расположена. Каждый массив имеет имя. Имена массивов имеют те же ограничения, что и имена переменных. В скобках после имени массива указываются индексы, т. е. написание массива выглядит следующим образом: `laArr[1,1]`. Индекс массива начинается с 1. Массивы бывают одномерными и многомерными (один столбец или несколько). Скобки возможны как круглые, так и квадратные, но обычно применяют квадратные, чтобы было легко отличить массив от функции. Массив в программе, как правило, нужно объ-

явить. Объявление массива производится командой DECLARE, которая имеет следующий синтаксис:

```
DECLARE массив1[количество_элементов1, количество_элементов2],
массив2[количество_элементов3, количество_элементов4]
```

и позволяет создать одно- или многомерный массив. Элементы массива могут иметь любой тип данных. Например, в двумерном массиве один столбец может хранить фамилии (т. е. символьные данные), а второй — даты рождения (данные типа дата). Сразу после объявления все элементы массива имеют логический тип и значение .F..

Массивы, создаваемые в командном окне, получают атрибут PUBLIC, а создаваемые программно — PRIVATE. Размерность массива во время работы программы может быть изменена.

Порядок следования элементов массива в памяти можно записать как laArr[1,1], laArr[1,2], laArr[1,3], laArr[2,1], laArr[2,2], laArr[2,3], т. е. данные хранятся списком. Поэтому можно обращаться к элементам этого списка по порядку: например, к элементу laArr[2,3] можно обратиться как к laArr[6].

Кроме команды DECLARE, объявить массив можно командой DIMENSION, которая очень похожа на DECLARE:

```
DIMENSION массив1[количество_элементов1,количество_элементов2]...
```

Массив, как и переменная, имеет область видимости, и может быть локальным (LOCAL), личным (PRIVATE) и общим (PUBLIC).

Для объявления общих (или глобальных) массивов служит команда PUBLIC:

```
PUBLIC массив1[количество_элементов1, количество_элементов2]
```

Для объявления локальных массивов используется команда LOCAL:

```
LOCAL массив1[количество_элементов1, количество_элементов2].
```

Операции с массивами в FoxPro характеризуются очень высокой скоростью исполнения, поэтому они так популярны у программистов. Для работы с массивами можно использовать команды APPEND FROM ARRAY, COPY TO ARRAY, SCATTER, GATHER. Следует помнить о том, что команда COPY TO ARRAY игнорирует memo-поля, для работы с memo-полями существуют свои собственные команды.

Процедуры и функции

Программный файл может содержать процедуры и функции, хотя они могут располагаться и в отдельных файлах (тоже с расширением prg). Процедурный файл — это отдельный фрагмент кода, который можно вызывать из разных мест программы. Если процедуры находятся в программном файле, они должны располагаться за последней строкой программного кода. Оформляется процедура заголовком PROCEDURE, а заканчивается ENDPROC или RETURN. Процедура может иметь синтаксис двух видов:

```
PROCEDURE ProcedureName
   [ LPARAMETERS parameter1 [ ,parameter2 ] ,... ]
      Commands
   [ RETURN [ eExpression ] ]
[ENDPROC]
```

или

```
PROCEDURE ProcedureName( [ parameter1 [ AS para1type ][ ,parameter2
[AS para2type ] ] ,...] ) [ AS returntype ]
      Commands
   [ RETURN [ eExpression ] ]
[ENDPROC]
```

Команда LPARAMETERS присваивает локальным переменным памяти или массивам переменных значения, переданные из вызывающей программы.

Синтаксис

```
LPARAMETERS ParameterList
```

Параметры

ParameterList

Задает имена локальных переменных памяти или массивов, которым присваиваются данные.

Параметры в списке ParameterList разделяются запятыми. В операторе LPARAMETERS должно присутствовать по крайней мере столько же параметров, сколько и в операторе DO ... WITH. Если в операторе LPARAMETERS окажется больше переменных или массивов, чем передается оператором DO ... WITH, лишние переменные или массивы инициализируются значением "ложь" (.F.). Всего может быть передано не более 26 параметров.

Чтобы определить число параметров, переданных в последнюю выполнявшуюся программу, процедуру или пользовательскую функцию, можно использовать функцию PCOUNT() или PARAMETERS(), причем PCOUNT() более предпочтительна.

Если необходимо обойти ограничение в 26 передаваемых параметров, можно передавать их по ссылке (Do MyProc With ...) — в этом случае количество параметров не ограничено.

LPARAMETERS должен быть первым исполняемым оператором вызываемой программы, процедуры или пользовательской функции, если вы передаете в эту программу значения, переменные или массивы.

По умолчанию оператор DO ... WITH передает переменные и массивы в процедуры по ссылке. Когда в вызываемой процедуре значение изменяется, новое значение передается назад в соответствующую переменную или массив вызывающей программы. Если вы хотите передать переменную или массив в процедуру по значению, заключите эту переменную или массив в скобки в списке параметров DO ... WITH. Никакие

изменения параметра в вызываемой процедуре не будут передаваться обратно в вызывающую программу.

По умолчанию переменные передаются в процедуру по ссылке и в пользовательскую функцию по значению. Для передачи переменных в пользовательскую функцию по ссылке пользуйтесь командой SET UDFPARMS REFERENCE.

Если параметры принимаются через команду PARAMETERS, то область видимости PRIVATE. А если через команду LPARAMETRS, то область видимости LOCAL. Процедура может принимать при ее вызове параметры от вызывающей программы. Параметры могут объявляться как частные переменные (ключевое слово PARAMETERS) или как локальные переменные (ключевое слово LPARAMETERS). Как именно объявлять параметры — выбор программиста. Число передаваемых параметров может быть меньше, чем число объявленных параметров. В этом случае "лишние" параметры процедуры инициализируются в значение False (.F.).

Функция, как и процедура, является подпрограммой и тоже имеет синтаксис двух видов:

```
FUNCTION FunctionName
   [ LPARAMETERS parameter1 [ ,parameter2 ] , ... ]
      Commands
   [ RETURN [ eExpression ] ]
[ENDFUNC]
```

или

```
FUNCTION FunctionName( [ parameter1 [ AS para1type ][ ,parameter2
[ AS para2type ] ],...] ) [ AS returntype ]
      Commands
   [ RETURN [ eExpression ] ]
[ENDFUNC]
Конец формы
```

Можно вызывать процедуру как

```
DO <имя процедуры> IN <программный модуль>,
```

а можно, как и функцию, достаточно ввести ее имя с парой скобок, например, CleanUp().

Кроме встроенных функций, могут быть функции, разработанные пользователем.

Они называются UDF (User Defined Functions).

Поиск функций и процедур происходит по следующей цепочке: текущая программа | отдельный процедурный файл (определяемый SET PROCEDURE TO) | текущий каталог | поиск в пути указанном командой SET PATH TO.

В Visual FoxPro существует большое количество встроенных функций для обработки различных видов переменных и массивов. Эти функции подразделяются на:

- математические;

- битовые;
- статистические;
- символьные;
- функции даты и времени;
- логические;
- функции управления данными;
- функции управления индексами;
- функции управления записями;
- функции установления отношений между таблицами;
- функции управления курсорами, таблицами и базами данных;
- функции управления средой окружения;
- функции управления файлами и папками;
- функции управления меню;
- функции управления печатью и отчетами;
- функции работы с клавиатурой и мышью;
- функции управления окнами;
- функции для работы в сети;
- функции управления массивами;
- функции для обработки ошибок;
- функции управления событиями;
- функции для работы с объектами;
- DDE-функции;
- SYS-функции.

Список функций вы найдете в *приложении 3* и на CD-диске к книге: CHAR1\spravka \spravka.exe. Большинство функций интуитивно понятны и не требуют дополнительных пояснений. В качестве примера приведены несколько математических и символьных функций.

Математические, числовые, статистические и битовые функции

Поскольку FoxPro — это язык работы с базами данных, он имеет вполне достаточное количество функций для работы с числами, статистикой и математикой (табл. 1.6).

Таблица 1.6. *Функции языка FoxPro*

Функция	Описание
ABS()	Возвращает абсолютное значение заданного числового выражения
CALCULATE	Функция позволяет выполнять операции как над полями таблицы, так и с арифметическими выражениями
CAST()	Определяет выражение данных, обычно в инструкции SQL, которое вы хотите преобразовать в другой тип данных. Выражение может быть полем, вычисляемым полем или другим типом выражения
INT()	Возвращает целую часть числа
VAL()	Возвращает числовое или денежное значение из символьного выражения, состоящего из чисел

Символьные функции

Одна из замечательных особенностей FoxPro состоит в огромном количестве разнообразных функций для обработки символьных данных (табл. 1.7).

Таблица 1.7. *Символьные функции*

Функция	Описание
$	Возвращает значение "истина" (.T.), если данное символьное выражение содержится в другом символьном выражении, в противном случае возвращает "ложь" (.F.)
ALLTRIM	Удаляет все ведущие и концевые пробелы или символы синтаксического анализа из указанного символьного выражения, или все ведущие и концевые нулевые байты из указанного двоичного выражения
AT	Ищет в символьном выражении вхождение другого символьного выражения. Поиск зависим от регистра
ATC	Возвращает начальную позицию первого вхождения символьного выражения или memo-поля в другое символьное выражение или memo-поле, независимо от регистра этих двух выражений
ATLINE	Возвращает номер строки с первым вхождением символьного выражения или memo-поля в другое символьное выражение или memo-поле, считая от первой строки
ATCLINE	Возвращает номер строки с первым вхождением символьного выражения или memo-поля, независимо от регистра символов в обоих выражениях (т. е. прописные или строчные)
BETWEEN	Определяет, находится ли заданное некоторое выражение между значениями двух заданных выражений того же типа данных
LEFT	Возвращает из символьного выражения заданное число символов, начиная с самого левого
LOWER	Возвращает заданное символьное выражение в нижнем регистре (маленькими буквами)

LTRIM	Удаляет все ведущие пробелы или символы синтаксического анализа из указанного символьного выражения или все ведущие нулевые байты указанного двоичного выражения
OCCURS	Возвращает количество вхождений, когда одно символьное выражение "встречается" (входит) в другое символьное выражение
PROPER	Преобразует символьное заданное выражение в строку с заглавными первыми символами
RAT	Возвращает числовую позицию последнего вхождения символьного выражения в другое символьное выражение
RATLINE	Возвращает номер строки последнего вхождения символьного выражения в другое символьное выражение или memo-поле, считая с последней строки
RIGHT	Возвращает указанное количество самых правых символов из символьной строки
RTRIM	Удаляет все концевые пробелы или символы синтаксического анализа из указанного символьного выражения или все концевые нулевые байты из указанного двоичного выражения
SUBSTR	Возвращает подстроку из заданного символьного выражения или memo-поля, начиная с указанной позиции в этом символьном выражении, или memo-поле и содержащую указанное количество символов

Таблица 1.7 *(окончание)*

Функция	**Описание**
TRIM	Удаляет все конечные пробелы или символы синтаксического анализа из определенного символьного выражения, или все конечные нулевые (0) байты из определенного двоичного выражения
UPPER	Возвращает заданное символьное выражение, преобразованное в верхний регистр (заглавными буквами)

Выражения и операторы

Выражения

Выражением называют какой-либо из элементов языка или их совокупность, которая в результате обработки приводит к одному определенному значению. Выражения могут содержать однотипные переменные, функции, имена полей и константы, объединяемые знаками операций.

Выражения бывают:

- числовые;
- символьные;
- типа даты и времени;
- логические.

В числовых выражениях используются следующие знаки операций:

+, -, *, / — основные арифметические действия;

% — остаток от деления;

** или ^ — возведение в степень.

В символьных выражениях используется конкатенация, т. е. сложение строк, например "Строка1"+ "Строка2".

При использовании в символьных выражениях знака "-" хвостовые пробелы первой строки перемещаются в конец строки-результата.

Например:

```
Stroka1="aaa   "
Stroka2="bbb   "
Rez=Stroka1-Stroka2
? rez
"aaabbb      "
```

Немного о макроподстановках

Макроподстановка — это команда, имеющая синтаксис

```
 & VarName[.cExpression],
```

где VarName — имя переменной или массива. Макроподстановка используется для простейшей замены: во всех местах, где встречается VarName, вместо него будет помещен *замещающий текст*. При написании VarName нельзя использовать префикс m. — будет сгенерирована синтаксическая ошибка. Кроме того, в циклах DO WHILE , FOR, SCAN вычисление макроподстановки происходит только при старте цикла.

Пример:

```
REPORT FORM &_REP NOEJECT NOCONSOLE TO PRINTER PROMPT PREVIEW
&_REP будет заменено при выполнении на имя отчета.
```

Максимальная длина подстановки 8192 символа. Для того чтобы обойти это ограничение, можно пользоваться составными макроподстановками:

```
Lca="SELECT"
Lca1=Lca+"* from table1 where table1.priznak=.t. into cursor cursor1"
&Lca1
```

Там, где это возможно, используйте выражение имени вместо макроподстановки. Выражение имени действует подобно макроподстановке, но обрабатывается быстрее.

Примеры правильного и неправильного применения макроподстановок:

```
lcFile1="C:\Program Files\My Super Program\Table1.DBF"
lcFile2="C:\Program Files\My Second Super  Program\Table1.DBF"
Copy File &lcFile2 to &lcFile2
```

Так как в пути пробелы — ошибка обеспечена.

А вот так будет правильно:

```
Copy File (lcFile1) to (lcFile2)
```

Аналогично происходит с командой `USE`:

`USE &lcFile1 &&` — ошибка;

`USE (lcFile1) &&` — нормально.

Операторы

Операторы бывают:

- арифметические;
- логические;
- сравнения;
- операторы действий с символьными строками.

Арифметические операторы (табл. 1.8) представляют собой знаки арифметических действий и позволяют выполнять действия над числами или переменными типа дата-время.

При использовании знака "+" для числовых переменных результатом будет сумма чисел. Если сложить символьные переменные, результатом будет сцепление строк.

Таблица 1.8. *Арифметические операторы*

Оператор	Описание
`()`	Повышение приоритета операции: действия в скобках выполняются первыми
`**` или `^`	Возведение в степень
`*`	Умножение
`/`	Деление
`%`	Модуль числа (остаток от деления)
`+`	Сложение
`-`	Вычитание

Сложение двух переменных типа дата невозможно. Возможно сложение типа `c={^2005/01/01}+10`, т. е. дата и какое-то число.

Порядок выполнения операций соответствует их расположению в табл. 1.9.

Таблица 1.9. *Логические операторы*

Оператор	Описание
`()`	Повышение приоритета операции: операции в скобках выполняются первыми

.NOT. или !	Логическое отрицание (НЕ)
.AND.	Логическое умножение (И)
.OR.	Логическое исключение (ИЛИ)

Если несколько выражений объединены оператором .AND., то результат будет =.T. в случае выполнения всех составляющих такого условия. Если хотя бы одно условие не будет выполнено, результат будет .F..

Если используется оператор ! или .NOT., это означает, что результат будет истинным в случае невыполнения условия. Например, X ! Y будет верным во всех случаях, когда X не равно Y.

Оператор OR возвратит истину (.T.), если хотя бы одно из условий, входящих в состав объединенного выражения, будет верным. Например, x=15 OR y=3 — выражение будет верно в случае, если либо x=15, либо y=3.

Таблица 1.10. *Операторы сравнения*

Оператор	Описание
=	Равно
<>, !=	Не равно
<	Меньше

Таблица 1.10 *(окончание)*

Оператор	Описание
>	Больше
<=	Меньше или равно
>=	Больше или равно
==	Точное равенство (полное совпадение)

Операторы сравнения (табл. 1.10) используются для сравнения выражений, при этом результат сравнения получает логическое значение .T. или .F.. Сравнивать можно только выражения одного типа: числовые с числовыми, символьные с символьными, даты с датами. В противном случае получим тривиальную ошибку "Operator/operand type mismatch".

Пример:

```
lcX=125
                lcY="125"
```

Здесь нельзя сравнивать lcX=lcY.

Можно вести речь о сравнении только lcX=val(lcY).

Управляющие конструкции

Команды в программе выполняются в порядке их написания. А как же быть в случае, если порядок выполнения команд нужно изменить? Для этого в FoxPro предусмотрен ряд управляющих конструкций. К таким управляющим конструкциям языка относятся:

- команды и функции ветвления:
  - `IF ... ENDIF`
  - `IIF()`
  - `CASE ... ENDCASE`
  - `ICASE()`
- циклы:
  - `SCAN ...ENDSCAN`
  - `FOR ... ENDFOR`
  - `WHILE ... ENDWHILE`
- команды выхода или безусловного перехода:
  - `EXIT`
  - `DO`
  - `RETURN`
  - `QUIT`

Условный переход `IF ... ENDIF` используется для выбора одного из двух вариантов и имеет следующий синтаксис:

```
IF <УСЛОВИЕ>
    <ДЕЙСТВИЯ, ВЫПОЛНЯЕМЫЕ В СЛУЧАЕ ВЫПОЛНЕНИЯ УСЛОВИЯ>
ELSE
   <ДЕЙСТВИЯ, ВЫПОЛНЯЕМЫЕ В СЛУЧАЕ НЕВЫПОЛНЕНИЯ УСЛОВИЯ>
ENDIF
```

При этом оператор `ELSE` не является обязательным и может быть опущен. Разрешаются и вложения операторов `IF`, однако они могут усложнить программу настолько, что вы сами с трудом в ней будете разбираться. В этом случае всегда применяйте комментарии.

Комментарии — это ваши пояснения к программе. Они не будут восприняты FoxPro в качестве команд, если вы сделаете пометку в начале строки комментариев в виде знака `*` или оператора `NOTE`. Если же комментарий располагается прямо в строке с командой, отделите комментарии двойным амперсандом (`&&`). В этом случае все, что находится за амперсандом, будет считаться комментарием.

Функцию `IIF()` удобно применять там, где требуется одна строка условия: в отчетах, этикетках, формах и т. д. Выглядит она так:

```
IIF(lExpression, eExpression1, eExpression2),
```

где:

`lExpression` — анализируемое логическое выражение;

`eExpression1` — значение параметра, если условие выполнено;

`eExpression2` — значение параметра, если условие не выполнено;

Так же, как и `IF ... ENDIF`, `IIF()` может иметь вложенности. Пример:

```
IIF(vxod.srok_isp-DATE()=1 .And. prizn_isp=.F. and !empty(vxod.srok_isp),
"Последний день!", IIF(srok_isp<DATE() .And. prizn_isp=.F. and
!empty(vxod.srok_isp),  "Просрочено", ""))
```

Обращаться с вложенными `IIF()` следует с особой аккуратностью.

Если условий много, к тому же они взаимоисключающие, то лучше использовать конструкцию `CASE ... ENDCASE`. Синтаксис конструкции имеет вид:

```
DO CASE
CASE <УСЛОВИЕ1>
       <действия, выполняемые в случае выполнения условия1>
CASE <УСЛОВИЕ2>
       <действия, выполняемые в случае выполнения условия2>
               OTHERWISE
                       <действия, выполняемые в случае невыполнения всех
предыдущих
                        условий>
ENDCASE
```

Условие, вероятность выполнения которого наиболее велика, помещается вверху списка условий.

Пример:

```
DO CASE
        CASE lcn<20
                lcn=0
        CASE lcn<25
          lcn=20
        OTHERWISE
                lcn=1.25*lcn
ENDCASE
```

Операторы `IF ... ENDIF` и `DO CASE ... ENDCASE` можно сочетать вместе. Обе структуры допускают вложенность. Пример:

```
if Between(ncount,1,3)
   do case
      case ncount=1
        cErrorMsg='Изменился счетчик 1'
               case ncount=2
```

```
                    cErrorMsg='Изменился счетчик 2'
            case nStat=3
                    cErrorMsg='Изменился счетчик 3'
        endcase
endif
```

Удобной и полезной управляющей функцией является `ICASE()`, она имеет следующий синтаксис:

```
ICASE( <Логическое условие1>, <Оценка результата1> [, <Логическое условие2>,
<Оценка результата2>] ...
```

Пример применения функции `ICASE`:

```
lcvar= ICASE(1+2=2,"First is false",1+1=3,"Second is false", "None are true")
```

Переменной `lcvar` будет присвоено значение `None are true`, поскольку первое и второе условия оцениваются как `.F.`.

Циклические конструкции

Циклические конструкции позволяют повторять некоторую часть программного кода, называемую телом цикла, некоторое необходимое количество раз.

В цикле

```
DO WHILE <условие>
   <тело цикла>
ENDDO
```

`<тело цикла>` выполняется, пока `<условие>` является истинным. Если `<условие>` получило значение `.F.`, выполняется команда, следующая за `ENDDO`.

Пример:

```
DO WHILE ! FEOF(handle)
        =FGET(handle)
ENDDO
```

В этом примере операция чтения открытого файла будет продолжаться до тех пор, пока не будет обнаружен конец этого файла.

Циклическая конструкция `SCAN ... ENDSCAN` используется, как правило, для целого файла таблицы базы данных. Эта конструкция может употребляться с условием `(FOR, WHILE)`, а может и без условия, в этом случае сканируется вся таблица.

Циклическая конструкция `FOR ... ENDFOR` служит для выполнения цикла определенное количество раз. Выглядит она следующим образом:

```
FOR <счетчик – константа, переменная или выражение> [FROM <начальное значение>]
[TO <конечное значение>] [STEP <шаг>]
        <тело цикла>
ENDFOR
```

Пример:

```
 FOR I=1 TO 10 STEP 2
          A(I)=A(I)+lcCount
  ENDFOR
```

Ключевое слово STEP можно опустить, в этом случае шаг считается равным 1.

Шаг может быть также отрицательной величиной, тогда начальное значение (FROM) должно быть больше конечного.

При работе с циклическими конструкциями нужно избегать неисполнимых и бесконечных циклов. Цикл становится неисполнимым, если начальное значение больше конечного и шаг положителен, например:

```
FOR I=10 TO 1 STEP 1
    nlCount=nlCount+3
ENDFOR
```

Бесконечным будет следующий цикл:

```
FOR I=1 to 10
   I=1
ENDFOR
```

Директивы препроцессора

Константам можно присваивать значения не только во время исполнения, но и во время компиляции, при этом подстановка значений производится во время компиляции, что дает некоторую экономию памяти и, вероятно, увеличивает быстродействие. Присвоение производится директивой #DEFINE:

```
#DEFINE True .t.
#DEFINE False .f.
```

и помещается обычно в начало файла.

Для добавления в другие программы используется директива #INCLUDE, например:

```
#INCLUDE Excel.h
```

Зарезервированные слова

Зарезервированные слова в Visual FoxPro включают функции, системные переменные, свойства, события, методы, команды, константы меню и предложения. При программировании избегайте использования резервных слов в качестве названия окна, таблицы или имени области. Если вы используете резервное слово как имя, это может вызвать синтаксическую ошибку. Список зарезервированных слов вы найдете на CD в справочной программе Spravka.

Системные переменные

Системные переменные являются встроенными переменными, которые Visual FoxPro создает и поддерживает автоматически. Они определены как PUBLIC по умолчанию, но вы можете объявить их как PRIVATE. Список системных переменных можно найти в упоминавшейся ранее справочной системе: CHAR1\spravka\spravka.exe.

Таблицы, курсоры и базы данных

Базой данных в FoxPro называется контейнер, хранящий таблицы, индексы, отношения, представления, процедуры и функции. Сама база данных имеет расширение dbc, а ее поля типа `Memo` — dct, и еще имеется файл индексов — dcx. Для создания базы данных, изменения ее структуры и выборки данных используются системы управления базами данных.

Таблица — это файл DBF, а также связанные с ним файлы FPT (для хранения полей типа `Memo` и `General`), и файл CDX, так называемый структурный индексный файл. Таблицу можно представить как совокупность строк (записей) и столбцов (полей), имеющих уникальное имя в пределах одной базы данных.

Поле таблицы — это столбец**,** который вы видите каждый раз, открывая таблицу для просмотра. Поле таблицы, включенной в базу данных, может содержать до 128 символов, содержать русские символы, цифры и некоторые специальные символы.

Таблицы в Visual FoxPro могут существовать как отдельные, свободные, так и в составе баз данных. Каждая таблица, включаемая в состав базы данных, принадлежит только одной базе. При необходимости использовать уже используемую таблицу в другой базе данных ее можно скопировать под другим именем:

```
OPEN DATABASE new
USE table1
COPY to c:\new
```

Каждая таблица при создании получает имя, по умолчанию это имя `Table1.dbf`, оно может быть изменено. Имя таблицы — это имя файла, поэтому при именовании таблицы необходимо соблюдать требования операционной системы. Имена таблиц должны начинаться с буквы или значка подчеркивания (_) и могут содержать буквы, цифры и значки подчеркивания.

Типы табличных данных

Существуют типы данных, используемых только для полей в таблицах (табл. 1.11).

Таблица 1.11. *Типы данных полей в таблицах*

Тип данных	Описание
`Blob`	Двоичные данные неопределенной длины. Начинается обязательно с `0h`, например `0hABCD`. Размер 4 байта в таблице. Хранится в memo-поле (FPT-файл). Не имеет установленной кодовой страницы. Размер ограничивается либо размером памяти, либо действует ограничение на размер таблицы в 2 Гбайт
`Character`	Символьный тип. Может содержать буквы, цифры, знаки пунктуации, пробелы, спецсимволы, нулевые байты. Максимальный размер символьной строки составляет 16 777 184 символа

Character (Binary)	Любые символьные данные, для которых вы не хотите изменять кодовую страницу. Длина — от 1 до 254 символов
Currency	Денежный тип, представляет собой 8-байтное значение валюты, для него перед числом устанавливается знак $. Максимальный размер составляет от — $922337203685477.5807 до $922337203685477.5807
Date	Хронологический тип, дата, содержит дни, месяцы, годы. Размерность 8 байт, может содержать даты от {^0001-01-01} до {^9999-12-31}
DateTime	Данные хронологического типа, содержит дату (дни, месяцы, годы) и время (часы, минуты, секунды) Размерность 8 байт. Диапазон значений может быть от {^0001-01-01} до {^9999-12-31) для даты и от 00:00:00 до 23:59:59 для времени.
Double	Числовые данные с плавающей точкой двойной точности. Применяется для вычислений, требующих высокой точности (например, научных). Диапазон значений может включать от +/-4.94065645841247E-324 до +/-8.9884656743115E307. Длина 8 байт
Float	То же самое, что и Numeric. Длина от 8 байт в памяти и от 1 до 20 байт в таблице. Диапазон значений от – .9999999999E+19 до .9999999999E+20
General	Ссылка на OLE-объект, например, страницу в Microsoft Excel. Длина 4 байта в таблице, лимитируется только объемом памяти
Integer	Целое число без дробной части. Занимает 4 байта, диапазон разрешенных значений от -2147483647 до 2147483647
Integer (Autoinc)	То же самое, что и Integer, только с автоматически добавляемой единицей к каждому следующему значению. Тип только для чтения. Занимает 4 байта, значение контролируется автоматически
Logical	Логический тип, может принимать только два значения — "истина" (.T.) или "ложь" (.F.)

Таблица 1.11 *(окончание)*

Тип данных	Описание
Memo	Символьный текст неопределенной длины или ссылка на блок данных. Занимает 4 байта, лимитируется объемом памяти
Memo (Binary)	То же самое, что и Memo, но с неизменяемой кодовой страницей. Занимает в таблице 4 байта, лимитируется объемом памяти
Numeric	Числовой тип, может содержать только целые либо десятичные числа. Размерность — 8 байт в памяти и от 1 до 20 байт в таблице. Диапазон значений от -.9999999999E+19 до +.9999999999E+20

`Varbinary`	Это двоичные величины или литералы фиксированной длины, не заполненные дополнительными нулевыми (0) байтами. Двоичные литералы начинаются с префикса `0h`, сопровожденного строкой шестнадцатеричных чисел и не сопровождается кавычками, в отличие от символьных строк
`Varchar`	Символьный текст, занимает в таблице 1 байт на каждый символ, до 254 символов
`Varchar (Binary)`	То же самое, что и `Varchar`, но для случая, когда вы не хотите менять кодовую страницу. Занимает 1 байт на каждый символ, до 254 символов

Ключевые поля и индексация таблиц

Ключевое поле — это такое поле (или набор полей), однозначно идентифицирующее запись таблицы. Не может быть двух записей с одинаковым ключевым полем.

Внешний ключ — это поле таблицы, содержащее в себе значение ключевого поля другой таблицы. То есть просто ссылка на запись другой таблицы.

Для большинства задач в Visual FoxPro удобно использовать в качестве ключевого поля суррогатный ключ типа `Integer`. Ключевое поле желательно вводить для всех без исключения таблиц базы данных.

Естественный ключ — это поле или набор полей, которые имеют некий физический смысл. К примеру, это может быть табельный номер, номер паспорта и т. п.

Суррогатный ключ — это поле, которое не имеет никакого физического смысла и введено исключительно с целью однозначной идентификации записи. Информация, которая в нем содержится, никак логически не связана с информацией из прочих полей той же таблицы.

Ваши пользователи будут вносить записи в созданные вами таблицы. Трудно представить, что они будут вносить данные в каком-то порядке, скорее всего, это будет происходить по мере необходимости, и записи в таблице будут не упорядочены.

Индексы — это двоичные файлы, в которых номера записей определяются значением индекса. Для каждого индекса в индексном файле содержится уникальная ссылка, которая определяет именно эту запись, при этом поиск в таблице ведется не последовательным перебором записей, а обращением к нужной записи. Именно это делает поиск скоростным.

Индекс можно создавать по полю или по некоторой комбинации полей. Если, например, мы хотим упорядочить таблицу `SOTR` по номеру отдела и по фамилиям сотрудников внутри отдела, то мы должны создать такой индекс:

```
INDEX ON otdel+fio to OTDF
```

При использовании предложения `ASCENDING` записи упорядочиваются по возрастанию, а в случае использования `DESCENDING` — по убыванию. По умолчанию используется

ASCENDING. Если использовать фразу FOR — индексирование будет выполнено по условию. Например, можно включить в индекс не все отделы, а только те, номер которых больше 5. В этом случае команда INDEX будет выглядеть так:

```
INDEX ON otdel+fio to OTDF FOR val(otdel)>5
```

Индексы создаются на вкладке **Indexes** Конструктора таблиц.

Order, т. е. порядок индекса (Ascending или Descending), может быть изменен при активации кнопки с двунаправленной стрелкой, либо левой кнопкой мыши, либо клавишей <Spacebar>. Если стрелка направлена вверх, это возрастающий порядок индекса, если вниз — убывающий.

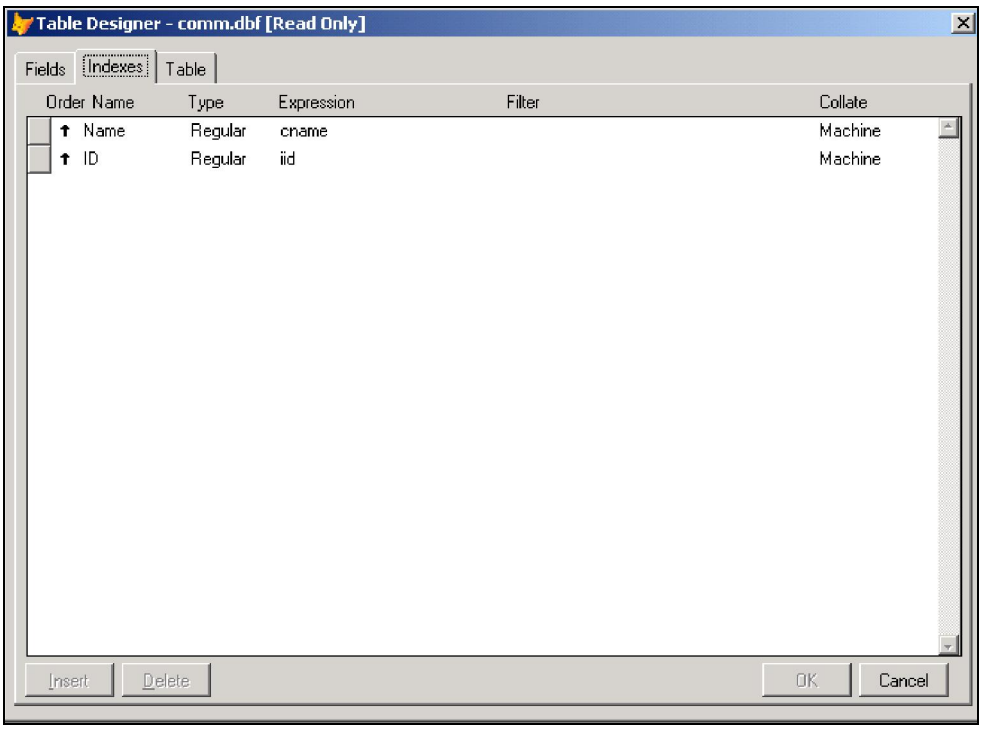

Рис. 1.1. Вкладка **Indexes** Конструктора таблиц

Name представляет собой имя индекса, или тега. Имя не может быть длиннее 10 символов и не может включать спецсимволы и пробелы.

Expression представляет собой имя поля или сложное выражение. Для формирования выражения можно воспользоваться кнопкой, находящейся справа от окна **Expression**.

Filter позволяет включать в индексное выражение некоторое логическое условие, при этом в индекс включаются не все записи, а только удовлетворяющие указанному условию.

`Collate`, установленное в состояние "`Mashine`", предполагает индексацию латинских символов. Если индексируемые символы — кириллица, нужно установить `Collate` в состояние `Russian`.

Индексы не обязательно должны быть уникальными, ведь цель их построения — наиболее быстрый доступ к записи. Список **Type** содержит все типы индексов, которые могут быть построены (табл. 1.12).

Таблица 1.12. *Типы индексов*

Типы индексов	Описание
`Primary` (первичный, уникальный)	Уникальный ключ, используется для организации отношений между таблицами и определения условий целостности данных. Не может иметь пустых и повторяющихся значений. Первичный ключ в таблице может быть только один
`Candidate` (потенциальный)	Это тоже уникальный ключ, который не может иметь пустых и повторяющихся значений. Фактически это то же `Primary`, а называется по-другому только потому, что таблица не может иметь несколько `Primary`
`Regular` (стандартный)	Обычный индекс, который может иметь повторяющиеся значения, содержит сам индекс и ссылку на запись
`Binary` (двоичный)	Это двоичный ключ, создающийся на основе логических выражений, например, при индексировании удаленных записей. Занимает мало места, не имеет нулевых значений

Поле `iid` таблицы `SOTR`, представляющее собой уникальный первичный ключ, имеет только неповторяющиеся значения, поскольку имеет тип `Integer (AutoInc)`, оно предназначено только для чтения и не подлежит изменению.

Индексация по сложным выражениям и использование хранимых процедур

В виде индекса может выступать не только поле, но и любая комбинация полей, а также, например, часть одного поля и часть другого. Для построения сложного выражения вызовите **Expression Builder**, нажав кнопку справа от окна **Expression**.

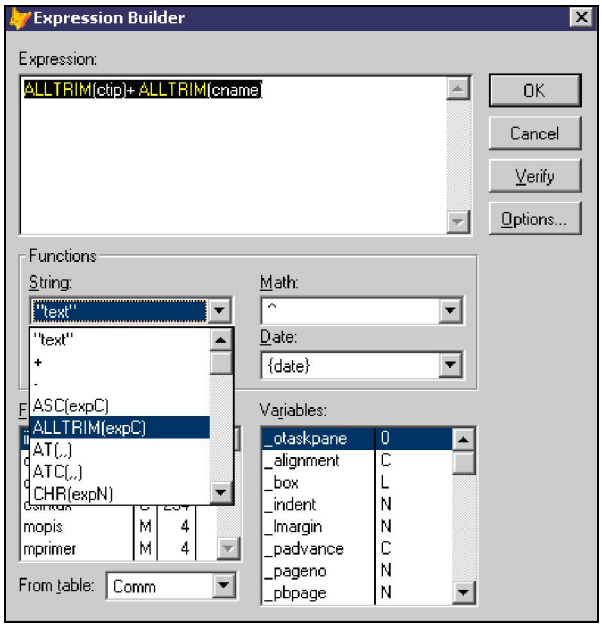

Рис. 1.2. Построение сложных индексных выражений с помощью **Expression Builder**

Разумеется, никто не может вам запретить заполнение поля **Expression** вручную, но можно воспользоваться и диалоговым окном. Диалоговое окно содержит множество функций для построения сложных выражений: символьные, математические, логические, функции для работы с датами. Сначала следует выбрать из раскрывающегося списка нужную функцию, а потом имя поля. В этом случае сразу произойдет замена вставки функции на нужное наименование поля. Поле можно выбрать не только из текущей таблицы. Нужную таблицу можно выбрать в окне **From table**. Правда, создавать индексы из нескольких таблиц — несколько рискованно. Иногда при индексировании таблицы применяют не стандартные, а разработанные самостоятельно функции. Ничего особенного в этом нет, пишут UDF-функцию (т. е. функцию, определенную пользователем), а в окне **Expression** указывают строку (UDF(поле)). Единственный недостаток этого способа — почему-то при индексации таблицы пользовательская функция всегда оказывается "не там", что вызывает ошибку. Теперь для сохранения пользовательских функций у нас имеется другой вариант: хранимые процедуры баз данных. Для того чтобы включить хранимую процедуру в состав базы данных, нужно:

1. Открыть проект.
2. Выбрать базу данных.
3. Найти группу "Хранимые процедуры" (Stored Procedures).
4. Нажать на кнопку **New**.

5. Написать процедуру, сохранить ее.
6. При создании индекса можно использовать написанную процедуру, указав, например, `Udf(dat_zakaz)`.

Отношения (связи) между таблицами

Любая реляционная база данных потому и называется реляционной, что характеризуется отношениями (relation) между таблицами. При этом одна таблица является родительской (главной), а вторая — дочерней (подчиненной). Чтобы установить отношение, используются индексы, которые синхронизируют перемещение указателя записи в связанных таблицах. Для определения отношений откроем окно Конструктора базы данных и произведем следующие действия:

1. Определяем родительскую таблицу.
2. Устанавливаем курсор на первичный ключ родительской таблицы.
3. Переместим курсор на индекс дочерней таблицы.
4. После того как мы отпустим кнопку мыши, образуется линия (связь), показывающая вновь созданное отношение. На стороне родительской таблицы отношение отображается одной линией, выходящей из таблицы, а на стороне дочерней таблицы — тремя линиями.

Мы можем отредактировать отношения, установив курсор мыши на линию, связывающую две таблицы, и дважды щелкнув на ней мышью. Появится окно редактирования отношения (рис. 1.3).

Слева мы видим информацию о родительской таблице, справа — о дочерней. Для того чтобы сохранить отредактированное отношение, нужно нажать кнопку **OK**, а чтобы отказаться — **Cancel**. Если отношение не нужно, его можно удалить, щелкнув правой кнопкой мыши. Линия связи при этом увеличится в толщине, и появится меню редактирования и удаления отношения, в котором достаточно выбрать пункт **Remove Relationship**, чтобы избавиться от ненужного отношения.

Второй способ удаления — выбрать отношение и нажать клавишу <Delete> на клавиатуре.

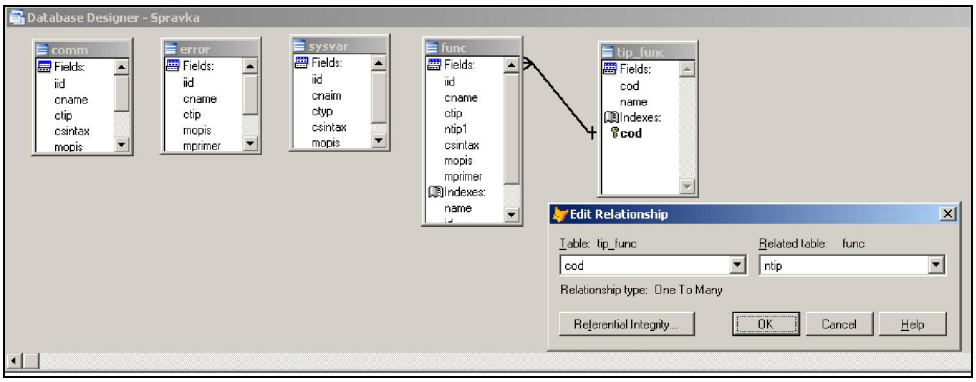

Рис. 1.3. Установка отношения (реляции)

Целостность данных при модификации связанных таблиц

Установленные отношения между таблицами можно использовать для установления условий проверки целостности данных. Что такое целостность данных?

Целостность в общем случае — это соответствие записей дочерней таблицы записям родительской, и наоборот. Приведем пример. Есть две таблицы: первая содержит информацию о лицах, взявших кредит в банке, вторая — о самих кредитах. Исчезновение записи в первой таблице приводит к тому, что невозможно определить лицо, взявшее кредит, а во второй — невозможно определить сумму и условия кредитования. И то и другое совершенно неприемлемо для банка. Поэтому при вводе, изменении и удалении записей следует определить правила для обеспечения целостности данных. До определения целостности базу данных нужно упаковать, применив команду **Database | Clean Up**. Затем надо выбрать из меню **Database** команду **Edit Referential Integrity** или правой кнопкой мыши щелкнуть на линии связи между таблицами и выбрать из контекстного меню пункт **Edit Referential Integrity**.

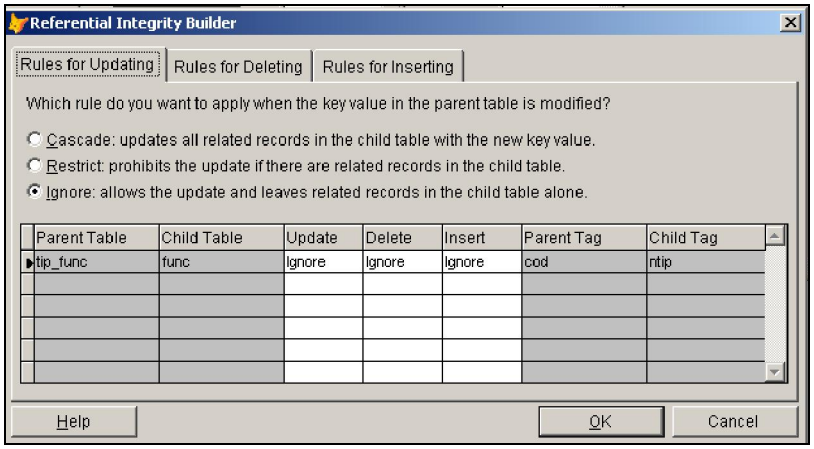

Рис. 1.4. Окно **Referential Integrity Builder**

В открывшемся окне (рис. 1.4) определяются правила обеспечения ссылочной целостности при изменении, удалении или добавлении записей. Для каждого из этих действий могут быть установлены следующие значения.

Правила для изменения записи в родительской таблице:

- **`Cascade`** — происходит каскадное изменение всех записей таблицы в соответствии с изменившимся значением ключа;
- **`Restrict`** — запрещается внесение изменений, если в дочерней таблице имеются связанные записи;
- **`Ignore`** — разрешаются любые изменения без изменения записей дочерней таблицы.

Правила для удаления записи в родительской таблице:

- **`Cascade`** — удаляются все связанные записи в дочерней таблице;
- **`Restrict`** — запрещается удаление записи родительской таблицы, если в дочерней таблице имеются связанные записи;
- **`Ignore`** — возможно удаление записи родительской таблицы без удаления записей дочерней таблицы.

Правила для добавления записи в родительскую таблицу или для модификации данных дочерней таблицы:

- **`Restrict`** — запрещается ввод новой записи в дочернюю таблицу, если в родительской нет для нее ключевого значения;
- **`Ignore`** — разрешается ввод новой записи.

По умолчанию, как видно, используются значения `Ignore`.

Триггеры и хранимые процедуры

При определении условий достоверности ввода записей используются триггеры и хранимые процедуры. *Триггеры* задают действия, выполняемые при добавлении, удалении или изменении записи. *Хранимые процедуры* содержат наиболее часто используемые процедуры, выполняемые сервером баз данных. Если вы определили условия достоверности ввода данных, их проверка осуществляется независимо от способа изменения данных в таблице.

Классы и объекты

Объектно-ориентированное программирование

Мы живем в мире объектов. Все, что мы видим — стол, дом, автомобиль, яблоко — все это объекты. Каждый объект имеет свои характеристики (свойства): стол, например, количество ножек, дом — количество дверей, автомобиль — мощность двигателя или цвет. Кроме характеристик, каждый объект имеет свои возможности (методы): автомобиль может ездить, яблоко можно съесть, в доме можно жить и т. д. Это даже не понятие программирования, а такое, общефилософское, понятие.

Итак, есть *объект* — это некая совокупность данных и функций, или, как говорилось выше, свойств и методов. Кроме того, существуют еще и события внешнего мира, на которые объект реагирует: например, вы завели автомобиль — он двигается. Вам, возможно, совсем неинтересно, что там происходит внутри двигателя, вас интересует только ваше воздействие и его результат — движение. *Событие* — это действие, направленное на изменение состояния объекта.

Теперь взгляните с точки зрения объектно-ориентированного программирования на само программирование. Есть, к примеру, объект — кнопка. У кнопки есть *свойства*, характеризующие объект — размер (Top, Width), цвет (ForeColor), название (Caption). Кроме того, кнопка обладает набором собственных *методов*, т. е. действий, которые объект способен выполнять. Например, мы знаем, что можно нажать на кнопку, т. е. существует метод `Click`. А что произойдет при нажатии кнопки: будет ли распечатан отчет, выполнены какие-либо вычисления или выбраны данные — это решает программист, описывая метод, который происходит при нажатии кнопки.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

- программа представляет собой модель некоторой части реального мира;
- модель реального мира или его части может быть описана как совокупность взаимодействующих между собой объектов;
- объект описывается набором параметров, значения которых определяют состояние объекта *(свойства)*, и набором операций *(методов)*, которые может выполнять объект;

- взаимодействие между объектами осуществляется посылкой специальных сообщений от одного объекта к другому. Сообщение, полученное объектом, может потребовать выполнения определенных действий, например, изменения состояния объекта *(событие)*;
- объекты, описанные одним и тем же набором параметров и способные выполнять один и тот же набор действий, представляют собой класс однотипных объектов.

Объектно-ориентированное программирование позволяет существенно сократить сроки разработки новых проектов и привязку разработанных проектов к новым условиям. Вместо того чтобы писать километры строк кода для определения поведения каждого объекта, ООП предоставляет в распоряжение разработчика прототипы объектов, которые легко настроить для своих собственных нужд.

Типы объектов

Классы и объекты тесно связаны друг с другом, хотя отождествлять их было бы неверным. Класс содержит информацию о том, как выглядит объект и какие он имеет свойства. Объект является экземпляром класса и наследует его характеристики. Большинство используемых классов являются видимыми (визуальными), а некоторые используются для объединения объектов или управления и не отображаются на экране (так называемые невизуальные классы). Соответственно, объект как экземпляр визуального класса тоже является *визуальным*. Объект, который может содержать внутри себя другие объекты, называется *контейнером.* Примером контейнера является форма, которая может содержать внутри себя сетку, кнопки, надписи и т. д.

Свойства объектов

Свойства (`Property`) — это данные об объекте, они определяют его внешний вид и поведение. Они привязаны к объекту и не существуют вне его. Для каждого объекта вы можете определить необходимый вам набор свойств, например, для формы можно указать размер, заголовок, цвет фона и букв текста, запретить ввод в некоторые поля, а в некоторых полях определить формат ввода данных. Для того чтобы получить доступ к свойствам объекта, достаточно выделить его, а затем вызвать контекстное меню правой кнопкой мыши. В появившемся меню выберите пункт **Properties**, и на экране появится окно свойств объекта. Свойства, представленные в списке, могут быть символьными, числовыми и логическими. Некоторые из них могут меняться произвольно, например, заголовки, другие меняют значения в определенных пределах. Каждое свойство объекта существует в программе до тех пор, пока существует сам объект. Свойства объекта могут быть добавлены или изменены. Возможность управлять свойствами объекта — это мощное средство управления данными. Объектно-ориентированное программирование имеет свой синтаксис, впрочем, довольно простой. Для того чтобы определить значение свойства, достаточно указать имя объекта и название свойства, разделив их точками:

```
Form1.Caption="Заголовок формы".
```

Язык Visual FoxPro позволяет устанавливать несколько свойств одного объекта, для этого используется оператор `WITH ... ENDWITH`:

```
WITH <имя объекта>
 [.имя свойства=выражение]
[.имя свойства=выражение]
ENDWITH
```

Например:

```
WITH thisform.command1
.ForeColor=RGB(192,192,192) && синий цвет фона
.BackColor=RGB(0,0,192)      && серый цвет символов
.Caption="Выполнить"
ENDWITH
```

(*См.* CD Char1\form2.scx).

Можно не только устанавливать свойства программным путем, но и добавлять их.

Для этого используется функция `ADDPROPERTY()`, имеющая следующий синтаксис:

```
ADDPROPERTY(oObjectName, cPropertyName, [, eNewValue ])
```

Например:

```
oMyForm = CREATEOBJECT('Form')
ADDPROPERTY(oMyForm, 'MyArray(2)', 1)
oMyForm.MyArray(2) = "Two"
CLEAR
? oMyForm.MyArray(1)
? oMyForm.MyArray(2)
```

События и методы

Кроме свойств, объекты обладают некоторыми возможностями, которые в объектно-ориентированном программировании называют методами. *Метод* — это то, что может делать объект. Например, автомобиль может ездить. Метод выполняется при наступлении некоторого события. В применении к программированию, при нажатии на кнопку мыши выполняется метод `Click()`. Именно методы обеспечивают обработку всех действий пользователя. При создании нового объекта, например, формы, список его методов пуст и выглядит так, как изображено на рис. 1.5.

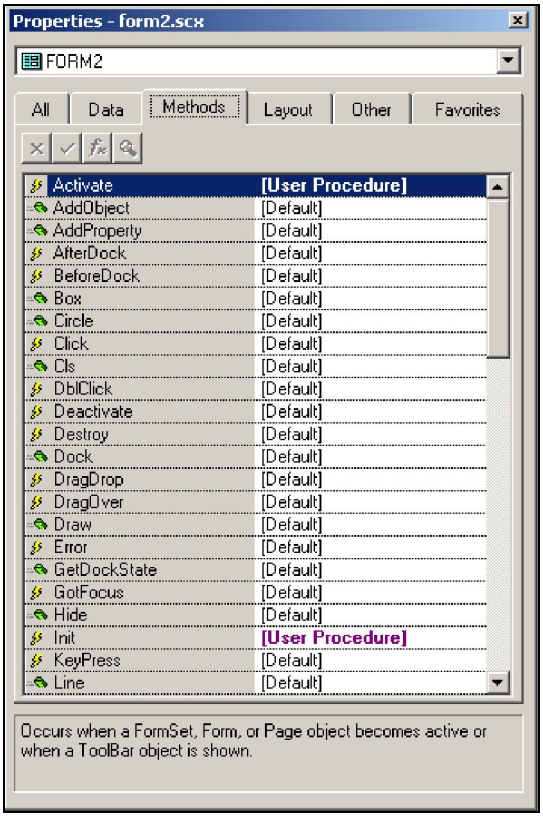

Рис. 1.5. Методы объекта "форма"

Пустые методы обозначены как `Default`.

Но мы всегда можем отредактировать эти значения, записав в метод тот код, который будет обрабатывать событие. Для редактирования метода нужно установить курсор на нужный метод в списке методов и дважды нажать кнопку мыши. При этом появляется окно (рис. 1.6), в котором нужно написать код, обрабатывающий событие. Например, для нажатой кнопки это может быть печать отчета или выполнение каких-либо вычислений. Чтобы не перегружать вас лишней информацией, в окне просто записан код вывода сообщения "`Выполнено`".

Список доступных свойств объектов, событий и методов с их кратким описанием приведен в *приложении 3*.

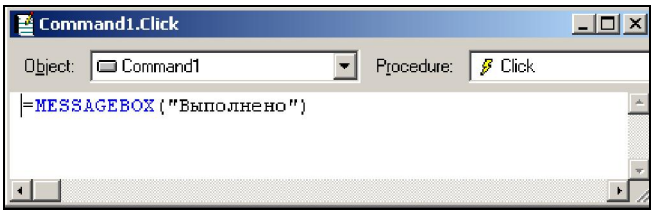

Рис. 1.6. Запись кода метода

В некоторых случаях требуется запретить выполнение действия, для этого используется команда NODEFAULT в начале метода, связанного с этим событием.

Для создания нового метода можно воспользоваться функцией RAISEEVENT(). Разумеется, можно создать новый метод и выполнив команду **Form | New Method.**

Для выполнения метода нужно указать не только сам метод, но и объект, к которому он относится.

В заключение хочется заметить, что свойства и методы добавлять можно, а события — нельзя.

Введение в классы Visual FoxPro

Объект — это экземпляр некоторого класса объектов или просто класса. Так автомобиль является экземпляром класса автомобилей, а лошадь — представителем класса животных. Класс — это абстрактное понятие, он содержит информацию о том, как должен выглядеть объект, и определяет выполняемые им действия. С точки зрения программирования, объектами являются практически все элементы интерфейса Visual FoxPro. Так, кнопка является представителем класса кнопок, а форма — представителем класса форм. Кнопка может быть нажата, а форма — открыта и закрыта, т. е. они имеют свой собственный набор событий и методов. В объектно-ориентированной программе программист не задумывается о прорисовке каждой формы или кнопки, вместо этого создается класс. Все объекты создаются как экземпляры соответствующего класса. Как же создать класс?

Для этого существует команда DEFINE CLASS. Синтаксис команды:

```
DEFINE CLASS ClassName1 AS ParentClass [OF ClassLibrary] [OLEPUBLIC]
   [[PROTECTED | HIDDEN] PropertyName1, PropertyName2 ...]
   [[.]Object.]PropertyName = eExpression ...]
   [PEMName_COMATTRIB = nFlags | DIMENSION PEMName_COMATTRIB[numElements]
      [PEMName_COMATTRIB[1] = nFlags
               PEMName_COMATTRIB[2] = cHelpString
               PEMName_COMATTRIB[3] = cPropertyCapitalization
               PEMName_COMATTRIB[4] = cPropertyType
               PEMName_COMATTRIB[5] = nOptionalParams]]
   [ADD OBJECT [PROTECTED] ObjectName AS ClassName2 [NOINIT] [WITH
cPropertylist]]
```

```
   [IMPLEMENTS cInterfaceName [EXCLUDE] IN TypeLib | TypeLibGUID | ProgID ]
   [[PROTECTED | HIDDEN] FUNCTION | PROCEDURE Name[_ACCESS |_ASSIGN]
      ([cParamName | cArrayName[] [AS Type][@]]) [AS Type]
      [HELPSTRING cHelpString] | THIS_ACCESS(cMemberName) [NODEFAULT]
            cStatements
   [ENDFUNC | ENDPROC]
ENDDEFINE
```

Приведем простой пример:

```
PUBLIC oForm
oForm=NEWOBJECT("Условия")
oForm.Show()
DEFINE CLASS Условия AS Form
    ADD OBJECT ogrid1 AS grid WITH RecordSource="MyCust",ColumnCount=1,;
    titlebar=0
    ADD OBJECT oButton1 AS CommandButton WITH TOP = 200

    PROCEDURE Load
      SELECT usl.naim FROM "usl" iNTO CURSOR MyCust

    PROCEDURE Init
        THISFORM.ogrid1.Column1.Width = 250
        THISFORM.ogrid1.Column1.FontSize=9
ENDDEFINE
```

В данном примере определяется форма с расположенным на ней объектом `Grid`, в `Grid` выводятся данные из курсора `MyCust`. Пример можно посмотреть на компакт-диске в Char6\primer_class.

Экземпляр класса создается с помощью функции `CREATEOBJECT()`.

```
oMyClass=CREATEOBJECT("Class").
```

В дальнейшем обращаться к свойствам объекта `oMyClass` можно так:

```
? oMyClass.Name
```

Базовые классы

При создании форм широко применяются базовые классы Visual FoxPro, список которых приведен в табл. 1.13.

Таблица 1.13. *Базовые классы Visual FoxPro*

Наименование	Описание	Видимый	Является контейнером
`ActiveDoc`	Активный документ	Нет	Нет
`CheckBox`	Флажок	Да	Нет

Таблица 1.13 *(окончание)*

Наименование	Описание	Видимый	Является контейнером
Column	Столбец	Да	Да
ComboBox	Раскрывающийся список	Да	Нет
CommandButton	Кнопка	Да	Нет
CommandGroup	Группа кнопок	Да	Да
Container	Контейнер	Да	Да
Control	Базовый визуальный класс	Да	Нет
Custom	Базовый невизуальный класс	Нет	Нет
EditBox	Область редактирования	Да	Нет
Form	Форма	Да	Да
FormSet	Набор форм	Нет	Да
Grid	Сетка	Да	Да
Header	Заголовок столбца сетки	Да	Нет
Hyperlink Object	Гиперссылка	Нет	Нет
Image	Изображение	Да	Нет
Label	Надпись	Да	Нет
Line	Линия	Да	Нет
ListBox	Список	Да	Нет
OLEContainerControl	OLE-объект управления		
OLEBoundControl	OLE-объект данных		
OptionButton	Переключатель	Да	Да
OptionGroup	Набор переключателей	Да	Да
Page	Страница формы	Да	Да
PageFrame	Набор страниц формы	Нет	Нет
ProjectHook	Проект	Нет	Нет
Separator	Разделитель	Да	Да
Shape	Обрамление	Да	Да
Spinner	Счетчик	Да	Да
TextBox	Поле ввода	Да	Да
Timer	Таймер	Нет	Нет

ToolBar	Панель управления	Да	Да

Базовые классы нельзя модифицировать, но их можно использовать для создания пользовательских классов.

Большинство классов являются видимыми, или визуальными, потому что отображаются в форме. Некоторые классы применяются для объединения объектов и не отображаются на экране. Кроме того, классы могут быть вложенными друг в друга. Класс, который может содержать другие классы, называется контейнером. Примером контейнера может являться обычная сетка (Grid). Мы можем получить доступ к любому ее элементу — от количества столбцов до надписи в заголовке колонки.

```
ThisForm. Grid1.ColumnCount=3
ThisForm.Grid1.Column1.Header1.caption="Количество"
```

Каждый базовый класс обладает минимальным набором свойств и событий (табл. 1.14 и 1.15).

Таблица 1.14. *Минимальный набор свойств базового класса*

Наименование	Описание
Class	Тип класса
BaseClass	Базовый класс, на основе которого создан данный класс
ClassLibrary	Библиотека классов, в которой хранится данный класс
ParentClass	Определенный пользователем класс, на основе которого был создан данный класс

Таблица 1.15. *Минимальный набор событий*

Наименование	Описание
Init	Наступает при создании объекта
Destroy	Наступает при освобождении объекта из памяти
Error	Наступает при возникновении ошибки, связанной с объектом

Таким образом, базовые классы (Base Class) — это базовые классы собственно FoxPro, т. е. элементы управления. Кроме Base Class, существуют Foundation Class — это набор классов, поставляемых вместе с FoxPro в папке FFC (Foundation File Classes), возможной основы для ваших собственных приложений. Base Foundation Class — это набор базовых классов для библиотек Foundation Class. Для большинства собранных там классов базовым является библиотека классов _base.vcx. Если открыть **Component Gallery**, то **Foundation Class** собраны в папке с одноименным названием, а **Base Foundation Class** собраны в папке с именем My Base Classes. По сути, эта папка — часть классов библиотеки классов _base.vcx, собственно, везде где

в описании указывается каталог вроде Visual FoxPro Catalog\Foundation Classes\Output\Report Listeners.

Подразумевается путь доступа в окне **Component Gallery**, а физически — папка FFC в основном каталоге FoxPro, а описание — это описание соответствующего класса. Его можно "разобрать на части" и посмотреть, как он внутри устроен, поскольку это не базовый класс FoxPro, а уже некий класс, написанный на основе базовых классов.

Характеристики ООП

Говорят, что все объектно-ориентированное программирование держится на трех "китах". На самом деле этих "китов" даже 4:

- инкапсуляция;
- наследование;
- подклассы;
- полиморфизм.

Эти термины многих пугают. Возможно, именно поэтому создалось расхожее мнение о немыслимой сложности объектно-ориентированного программирования, о том, что оно доступно только избранным. На самом деле все не так уж страшно.

Инкапсуляция

Инкапсуляция (encapsulation) — это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Это классическое определение инкапсуляции. Для чего она нужна? Для страховки от ошибок, ибо все мы люди и можем ошибаться. Инкапсуляция позволяет не напрямую обращаться к данным, а только лишь с помощью методов, что позволяет быстро выявить ошибку и исправить ее, а также предотвращает возможность *введения объекта в недопустимое состояние и/или несанкционированное разрушение этого объекта*.

Наследование и подклассы

Все объекты создаются на основе классов и наследуют свойства и методы классов.

Наследование (inheritance) — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов. Применение иерархии классов делает управляемыми большие потоки информации. Наследование приводит к повторному использованию однажды уже написанного кода. Наверное, не стоит говорить о том, как это важно для ускорения разработки программ. Наследование по определению заставляет что-то у чего-то наследо-

вать, так что вполне можно создать класс на основе другого класса. Такие классы называются *подклассами*. Для нового класса можно определить новые свойства и методы. Старые свойства и методы тоже никуда не деваются, если их специально не уничтожить.

Полиморфизм

При обычном процедурном программировании имя процедуры однозначно определяет выполняемый ею код. В объектно-ориентированном программировании вы можете использовать одни и те же имена методов для выполнения совершенно разных функций. *Полиморфизм* (polymorphism) — это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. Такой тип полиморфизма называется *перегрузкой операторов* (operator overloading).

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий. Преимущество полиморфизма в том, что он помогает снижать сложность программ.

Стандартные диалоги Visual FoxPro

Visual FoxPro содержит большое количество диалоговых окон для организации стандартных диалогов с пользователем. С помощью диалоговых окон можно строить выражения, создавать проект, устанавливать точки останова в отладчике, задавать новые методы и свойства, запускать программы на выполнение и т. д. Например, при необходимости открыть файл немедленно появляется диалоговое окно **Open**. Если вам нужно создать новый файл, введя в командном окне только одну команду `CREATE`, вы увидите окно с заголовком **Create**, в котором вы сможете выбрать нужный тип файла и определить имя для него. Открыть диалог можно как из главного меню Visual FoxPro, так и из командного окна или прямо из программы. Если вы используете для диалога главное меню, выберите **File | New**, и Visual FoxPro предложит вам диалоговое окно **New** (рис. 1.17).

В этом окне вы сможете создать проект, базу данных, таблицу и другие файлы.

Для ведения диалога из программ имеется целый ряд функций (табл. 1.16).

Таблица 1.16. *Функции*

Функция	Описание
AGETCLASS()	Показывает библиотеки классов в диалоговом окне открытия
GETCOLOR()	Отображает диалоговое окно **Color**
GETCP()	Запрашивает кодовую страницу в диалоговом окне

Таблица 1.16 *(окончание)*

Функция	Описание
GETDIR()	Отображает диалоговое окно **Select Directory**, в котором можно выбрать нужный каталог
GETFILE()	Отображает диалоговое окно **Open** и возвращает имя выбранного файла
GETFONT()	Отображает диалоговое окно **Font** и возвращает имя выбранного шрифта
GETPICT()	Открывает диалоговое окно **Open** и возвращает имя выбранного файла рисунка или графического изображения
GETPRINTER()	Открывает диалоговое окно выбора принтера. Возвращает имя выбранного принтера

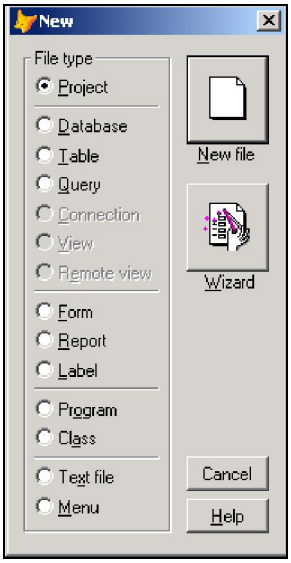

Рис. 1.7. Диалоговое окно **New**

Для вывода вспомогательной информации в контрольных точках программы или при обработке исключительных ситуаций можно использовать вызов диалога с нужным сообщением:

```
MessageBox('Ошибка!')
```

```
Синтаксис:
MESSAGEBOX(eMessageText [, nDialogBoxType ][, cTitleBarText][, nTimeout])
```

В качестве параметра диалога можно использовать произвольное строковое выражение, в том числе содержащее переносы на новые строки. Для переноса части сообщения на другую строку применяется `CHR(13)`. Если функции передано выражение, не являющееся строковым литералом, автоматически включается функция `TRANSFORM()`, чтобы преобразовать его в строковой эквивалент. Например, `Messagebox(date())` преобразует текущую дату в ее символьный эквивалент.