Language Reference

Functions, Statements, Methods, Properties, and Actions

Relational Database Management System for Windows™

This manual was produced using Microsoft Word for Windows.

Printed in Ireland: 03

Document Number DB26142-1092

Contents

Introduction v

Part 1 Language Summary

Part 2 A–Z Reference

Part 3 Appendixes

Appendix A ANSI Character Set 505

Appendix B Microsoft Access SQL 507

Index 515

Introduction

This manual is divided into three parts:

- Part 1, "Language Summary," provides a quick reference for looking up related functions, statements, methods, properties, and actions.
- Part 2, "A–Z Reference," includes descriptions of each function, statement, method, property, and action.
- Part 3, "Appendixes," contains the ANSI character set and describes Microsoft® Access SQL.

The information in this manual is the best available at the time of publication. In some cases, more up-to-date information may be available in online Help.

Document Conventions

This manual uses the following typographic conventions.

Example of convention	Description
Sub, If, ChDir, Print, Time	Words in bold with the initial letter capitalized are language-specific reserved words with special meaning to Access Basic.
FontSize, Item, SendKeys	Names of properties and actions appear with the initial letter capitalized. Concatenated names may contain other capital letters
expr, *path*	In syntax, text in italic indicates placeholders for information you supply.
[*color*]	In syntax, items inside square brackets are optional.
{**While** | **Until**}	In syntax, braces and a vertical bar indicate a mandatory choice between two or more items. You must choose one of the items unless all of the items also are enclosed in square brackets.
`Dim MyDB As Database` `Illegal function call`	This font is used for code and for error messages.

Example of convention	Description
V_DOUBLE C:\MYROOT\MYDIR\MYFILE.DOC	Words in all capital letters indicate constants or file names.
ENTER	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R. (Key names in online Help have only the initial letter capitalized.) The key names correspond to the names on the IBM Personal Computer keyboard. Other machines may use different names for keys.
ALT+F1	A plus sign (+) between key names indicates a combination of keys. For example, ALT+F1 means hold down the ALT key while pressing the F1 key.

`MyCaption = "This must be`
`➥ entered as one line`
`➥ in the Module window."`
	The line-continuation character (➥) indicates that code continued from one line to the next in the book should be typed all on one line in the Module window.

Programming Style

This manual follows the guidelines below in presenting programs for Access Basic.

- Reserved words are in lowercase with an initial capital letter; concatenated words may contain other capital letters. Constants appear in all capital letters:

```
If SomeValue = 0 Then TextValue$ = Str$(StartNumber%)
' If, Then, and Str$ are reserved words.

Global Constant PI = 3.141593
Circum = 2 * Radius * PI          ' Use constant for value of Pi.
```

- Variable names are in lowercase with an initial capital letter. Concatenated variable names may contain other capital letters:

```
Dim Income As Double
Dim MyFirstName As String * 30
```

- Line labels are used instead of line numbers for error-handling routines and for **On...GoSub** and **On...GoTo** statements:

```
On Error GoTo ErrorHandler
...
ErrorHandler
Print Error$
```

- An apostrophe (') introduces a comment:

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in **Sub** and **Function** procedures are indented:

```
Sub MyCode
  If X > 0 Then
     P = Q
  End If
End Sub
```

Language Summary

Functions, Statements, and Methods

This section contains four tables:

- Table 1 lists related functions and statements grouped by programming task.
- Table 2 lists methods grouped by programming task.
- Table 3 lists the functions that are available only in Access Basic.
- Table 4 lists the functions available in Microsoft Access expressions that are not available in Access Basic.

Table 1 Functions and Statements by Programming Task

Task type	Description	Function/statement
Arrays	Declare and initialize array variables	**Dim, Global, ReDim, Static**
	Reinitialize array variables	**Erase, ReDim**
	Find the limits of an array	**LBound, UBound**
	Change the default lower limit for arrays	**Option Base**
Control of program flow	Create loops	**Do...Loop, Exit Do, Exit For, For...Next, While...Wend**
	Handle errors	**On Error, Resume**
	Make decisions	**Choose, If...Then...Else, IIf, Select Case, Switch**
	Exit or pause the program	**DoEvents, End, Stop**
Conversion	ANSI value to string	**Chr**
	String to ANSI value	**Asc**
	Number to string	**Format, Str**
	String to number	**Val**
	One numeric data type to another	**CCur, CDbl, CInt, CLng, CSng, Fix, Int**
	Decimal number to other radix string	**Hex, Oct**
	Date to serial number	**DateSerial, DateValue**
	Serial number to date	**Day, Month, Now, Weekday, Year**

Table 1 Continued

Task type	Description	Function/statement
	Time to serial number	**TimeSerial, TimeValue**
	Serial number to time	**Hour, Minute, Now, Second**
Date/time	Get the current date or time	**Date, Now, Time**
	Set the date or time	**Date, Time**
	Time a process	**Timer**
	SQL dates	**DateAdd, DateDiff, DatePart**
Domain	Perform domain aggregate financial operations	**DAvg, DCount, DFirst, DLast, DMax, DMin, DStDev, DStDevP, DSum, DVar, DVarP**
Dynamic data exchange (DDE)	Use Access Basic as a DDE client	**DDEExecute, DDEInitiate**
	Use Access Basic to send data to a DDE server	**DDEPoke, DDESend**
	Use Access Basic to request data from a DDE server	**DDE, DDERequest**
	Terminate a DDE conversation	**DDETerminate, DDETerminateAll**
Error handling	Trap errors while a program is running	**On Error, Resume**
	Get error-status data	**Err, Erl**
	Get error messages	**Error**
	Simulate run-time errors	**Error**
Inspection of variables	Determine if a variable is a Date	**IsDate**
	Determine if a variable is Empty	**IsEmpty**
	Determine if a variable is **Null**	**IsNull**
	Determine if a variable is numeric	**IsNumeric**
	Determine the underlying data type of a **Variant**	**VarType**

Table 1 Continued

Task type	Description	Function/statement
File input/output	Access or create a file	**Open**
	Close a file	**Close, Reset**
	Write to a file	**Print #, Put, Write #**
	Control output appearance	**Spc, Tab, Width #**
	Read from a file	**Get, Input, Input #, Line Input #**
	Get information about a file	**EOF, FileAttr, FreeFile, Loc, LOF, Seek**
	Set or determine the read/write position in a file	**Seek**
	Manage disk drives or directories	**ChDrive, ChDir, CurDir, MkDir, RmDir**
	Manage files	**Dir, Kill, Lock...Unlock, Name**
Financial	Perform financial operations	**DDB, FV, IPmt, IRR, MIRR, NPer, NPV, Pmt, PPmt, PV, Rate, SLN, SYD**
Graphics on printed reports	Work with colors	**QBColor, RGB**
	Draw a shape	**Circle, Line, PSet**
	Find the size of text	**TextHeight, TextWidth**
	Change the coordinate system	**Scale**
	Print text	**Print**
Object manipulation	Display a dialog box	**InputBox, MsgBox**
Math	Perform general calculations	**Exp, Log, Sqr**
	Perform trigonometric functions	**Atn, Cos, Sin, Tan**
	Convert a number	**Fix, Int**
	Get the absolute value of a number	**Abs**
	Get the sign of an expression	**Sgn**
	Generate random numbers	**Randomize, Rnd**

Table 1 Continued

Task type	Description	Function/statement
Operators	Arithmetic	***, +, – , \ , /, ^, Mod**
	Comparison	**<, <=, >, >=, =, <>**
	Concatenation	**&**
	Logical	**And, Eqv, Imp, Not, Or, Xor**
	Pattern matching	**Like**
	SQL	**Is, In, Between...And**
Procedures	Define a procedure	**Function, Sub**
	Call a procedure	**Call**
	Exit a procedure	**Exit Function, Exit Sub**
	Declare a reference to an external procedure (DLL)	**Declare**
Recordsets		See Table 2
SQL	Perform an SQL aggregate financial operation	**Avg, Count, First, Last, Min, Max, StDev, StDevP, Sum, Var, VarP**
Strings	Format a string	**Format**
	Create a string of repeating characters	**Space, String**
	Work with ANSI values	**Asc, Chr**
	Manipulate a string	**InStr, Left, LTrim, Mid, Right, RTrim**
	Convert to lowercase or uppercase letters	**LCase, UCase**
	Left or right align a string	**LSet, RSet**
	Find the length of a string	**Len**
	Compare two strings	**StrComp**
Variables and constants	Declare a variable or constant	**Const, Dim, Global, Static**
	Set the default data type	**Def***type*

Table 1 Continued

Task type	Description	Function/statement
	Create a user-defined type	**Type**
	Assign a variable to a recordset object	**Set**
Miscellaneous	Run another program	**AppActivate, Shell**
	Handle events	**DoEvents**
	Generate keystrokes	**SendKeys**
	Get command-line arguments	**Command**
	Find environment variables	**Environ**
	Sound a beep	**Beep**
	Evaluate an expression	**Eval**
	Determine a number's position in a range	**Partition**

Table 2 Methods by Programming Task

Task type	Description	Method
OLE objects	Manipulate OLE objects	**AppendChunk, GetChunk**
Record manipulation	Add or delete a record	**AddNew, Delete**
	Make changes	**Edit**
	Save changes	**Update**
Record positioning	Move to a specific record	**FindFirst, FindLast, FindNext, FindPrevious, Seek**
	Move to a relative record	**MoveFirst, MoveLast, MoveNext, MovePrevious**
Recordset manipulation	Create a recordset	**Clone**
	Close a recordset	**Close**
	Return information about a recordset	**ListFields, ListIndexes, ListParameters, ListTables**

Table 3 Functions Available Only in Access Basic

Category	Description	Function
Arrays	Return the smallest or largest available subscript	**LBound**, **UBound**
Database manipulation	Refer to a database in code	**CurrentDB**, **OpenDatabase**
Finance	Return the rate of return for a series of periodic cash flows	**IRR**, **MIRR**
Printing	Determine the position for printing	**Spc**, **Tab**
Program execution	Suspend the execution of Access Basic	**DoEvents**

Table 4 Functions Used in Microsoft Access Expressions That Are Not Available in Access Basic

Category	Description	Function
DDE	Initiate a DDE conversation	**DDE**, **DDESend**
Field values	Perform statistical analysis	**Avg**, **StDev**, **StDevP**, **Var**, **VarP**
	Return the maximum or minimum value	**Max**, **Min**
	Return a value from the first or last record	**First**, **Last**
Records	Return the number of records	**Count**

Recordset Object Methods and Properties

This section contains two tables:

- Table 5 lists recordset objects and the methods that apply to each object.
- Table 6 lists recordset objects and the properties that apply to each object.

Table 5 Recordset Objects and Methods

Method	Dynaset	Snapshot	Table
AddNew	•		•
AppendChunk	•		•
Clone	•	•	•
Close	•	•	•
CreateDynaset	•	•*	•
CreateSnapshot	•	•*	•
Delete	•		•
Edit	•		•
FindFirst	•	•*	
FindLast	•	•*	
FindNext	•	•*	
FindPrevious	•	•*	
GetChunk	•	•	•
ListFields	•	•*	•
ListIndexes			•
MoveFirst	•	•	•
MoveLast	•	•	•
MoveNext	•	•	•
MovePrevious	•	•	•
Seek			•
Update	•		•

* Except **Snapshots** created with the **ListFields**, **ListIndexes**, **ListParameters**, or **ListTables** method.

Table 6 Recordset Objects and Properties

Property	Dynaset	Snapshot	Table
BOF	•	•	•
Bookmark	•	•	•
Bookmarkable	•	•	•
DateCreated			•
EOF	•	•*	•
Filter	•	•	
Index			•
LastUpdated			•
LockEdits	•		•
NoMatch	•	•	•
RecordCount	•	•	•
Sort	•	•*	
Transactions	•		•
Updatable	•		•

* Except **Snapshots** created with the **ListFields**, **ListIndexes**, **ListParameters**, or **ListTables** method.

Properties

The following table groups properties into functional categories.

Table 7 Properties by Programming Task

Category	Description	Property
Appearance	Hide or show an object or a section	Transparent, Visible
	Specify the color of an object or a section	BackColor, BackStyle, FillColor, ForeColor, Transparent
	Specify border appearance	BorderColor, BorderStyle, BorderWidth
	Create special effects	SpecialEffect
	Specify the direction of a line	LineSlant

Table 7 Continued

Category	Description	Property
Data source	Specify the source of a control's data	ControlSource
	Specify underlying records	RecordSource
	Specify the source for list entries	RowSource, RowSourceType
	Bind a list or combo box to a table field	BoundColumn
	Set the name of a bitmap to load in a command or toggle button	Picture
	Specify a numeric value for controls attached to an option group	OptionValue
	Restrict entry to a list	LimitToList
	Keep a running sum in a text box	RunningSum
Display	Specify views	DefaultView, ViewsAllowed
	Specify the display of data in an object frame or graph	Scaling
	Display gridlines on datasheets	ShowGrid
Editing	Enable editing	AllowEditing, AllowUpdating, DefaultEditing
	Specify whether data in a control can be changed	Locked
	Specify locking	LockEdits, RecordLocks
	Determine updating	Updatable, UpdateMethod
	Specify data entry	DefaultValue, ValidationRule, ValidationText
Form/report design	Specify when an object is displayed or printed	DisplayWhen
	Specify the size of a section or control	CanGrow, CanShrink, ColumnWidths, ListWidth

Table 7 Continued

Category	Description	Property
	Add a label when adding a control	AutoLabel
	Specify the number of columns or rows	ColumnCount, ListRows
	Specify column heads	ColumnHeads
	Specify the Cancel or default button	Cancel, Default
	Specify the status bar text	StatusBarText
	Display record selectors or scroll bars	RecordSelectors, ScrollBars
	Specify grid alignment units	GridX, GridY
Graphics	Resize to fit	AutoResize
	Draw	CurrentX, CurrentY, DrawMode, DrawStyle, DrawWidth, FillColor, FillStyle, ScaleHeight, ScaleLeft, ScaleMode, ScaleTop, ScaleWidth
Help	Specify the name or number of a Help topic	HelpContextID, HelpFile
Macros	Specify the name of a macro or user-defined function to run when an event occurs	AfterUpdate, BeforeUpdate, OnClose, OnCurrent, OnDblClick, OnDelete, OnEnter, OnExit, OnFormat, OnInsert, OnOpen, OnPrint, OnPush
	Specify whether a macro repeats	AutoRepeat
	Specify the name of a menu bar macro	OnMenu
Object references	Refer to an active object	ActiveControl, ActiveForm, ActiveReport
	Refer to a control	ControlName
	Refer to a section	Section
	Refer to a column in a combo or list box	Column
	Refer to a form or report	Form, FormName, Report

Table 7 Continued

Category	Description	Property
	Refer to a parent form or report	Parent
	Refer to a field on a parent form	LinkChildFields, LinkMasterFields
	Refer to underlying records	Bookmark, Dynaset
	Refer to a subform, subreport, or linked object	SourceObject
OLE objects	Describe the kind of OLE object	OLEClass
	Describe linked object data	Item
Printing	Count the number of times the current record in a form or report is formatted or printed	FormatCount, PrintCount
	Specify the printing of sections	ForceNewPage, KeepTogether, MoveLayout, NewRowOrCol, NextRecord, PageFooter, PageHeader, PrintSection
	Specify the printing of duplicate data	HideDuplicates
	Specify a page number	Page
Records	Count records	RecordCount
	Move between records	Bookmark, Bookmarkable
	Enable record filtering	AllowFilters
Recordsets	Sort records	Sort
	Indicate the beginning or end of a file	BOF, EOF
	Check for the success of a record find or seek	NoMatch
Size and position	Specify position	LabelX, LabelY, Left, Top
	Specify size	Height, Width

Table 7 Continued

Category	Description	Property
SQL	Specify selection criteria	Filter, SQL
Table design	Specify field attributes	DataType, Description, FieldName, FieldSize
	Specify the primary key	PrimaryKey
	Specify the table index	Index, Index1...Index5, Indexed
Text display	Specify text to display	Caption, StatusBarText
	Specify font	FontBold, FontItalic, FontName, FontSize, FontUnderline, FontWeight
	Specify text format	AddColon, DecimalPlaces, Format
	Align text	LabelAlign, TextAlign
Windows	Specify whether a form is modal	Modal
	Specify whether a form opens as a pop-up form	PopUp
	Get the handle for a form or report	hWnd
Miscellaneous	Enable a control to receive the focus	Enabled
	Determine the number of open forms or reports	Count
	Track creation or change dates for a table or query	DateCreated, LastUpdated

Actions

The following table groups actions into functional categories.

Table 8 Actions

Category	Description	Action
Data in forms and reports	Restrict data	ApplyFilter
	Move through data	FindNext, FindRecord, GoToControl, GoToPage, GoToRecord
Execution	Carry out a command	DoMenuItem
	Run a macro, procedure or query	OpenQuery, RunCode, RunMacro, RunSQL
	Run another application	RunApp
	Stop execution	CancelEvent, Quit, StopMacro, StopAllMacros
Import/export	Transfer data between Microsoft Access and other data formats	TransferDatabase, TransferSpreadsheet, TransferText
Object manipulation	Set the value of a field, control, or property	SetValue
	Update data or the screen	RepaintObject, Requery, ShowAllRecords
	Select a database object	SelectObject
	Copy or rename an object	CopyObject, Rename
	Open or close a database object	Close, DoMenuItem, OpenForm, OpenQuery, OpenReport, OpenTable
	Print a database object	OpenForm, OpenQuery, OpenReport, Print
	Move or resize a window	Maximize, Minimize, MoveSize, Restore
Miscellaneous	Display information on screen	Echo, Hourglass, MsgBox, SetWarnings
	Generate keystrokes	SendKeys
	Sound a beep	Beep
	Create a custom menu bar for a form	AddMenu

A-Z Reference

& Operator

Description Used to force string concatenation of two operands.

Syntax *result* = *operand1* **&** *operand2*

Remarks Whenever an operand is a number, it is converted to a **Variant** of **VarType** 8 (**String**). The data type of *result* is **String** if both operands are **String** expressions; otherwise, *result* is a **Variant** of **VarType** 8 (**String**). If both operands are **Null** (**VarType** 1), *result* is also **Null**. However, if only one operand is **Null**, that operand will be treated as a zero-length string when concatenated with the other operand. Any operand that is Empty (**VarType** 0) is also treated as a zero-length string.

See Also + Operator, Operator Precedence

Example This example uses the **&** operator to concatenate a number (`76`), a string literal that contains a space, and a string variable (`Var1`) that contains `"trombones"`. After concatenation, the result (`NewText`) contains `76 trombones`.

```
Var1 = "trombones"
NewText = 76 & " " & Var1
```

---

* Operator

Description Used to multiply two numbers.

Syntax *result* = *operand1* * *operand2*

Remarks The operands can be any numeric expression.

The data type of *result* is usually the same as that of the most precise operand. The order of precision, from least to most precise, is **Integer**, **Long**, **Single**, **Double**, **Currency**. The following are exceptions to this order:

- When multiplication involves a **Single** and a **Long**, the data type of *result* is converted to a **Double**.
- When the data type of *result* is a **Variant** of **VarType** 3 (**Long**), **VarType** 4 (**Single**), or **VarType** 7 (Date) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 5 (**Double**).
- When the data type of *result* is a **Variant** of **VarType** 2 (**Integer**) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 3 (**Long**).

If one or both operands are **Null** expressions, *result* is a **Null**. Any operand that is Empty (**VarType** 0) is treated as 0.

See Also + Operator, – Operator, / Operator, \ Operator, ^ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example determines total tax by using the * operator to multiply 10,000 by a tax rate of 18 percent. After the calculation, `TotalTax` equals 1800.

```
GrossPay = 10000                    ' Assign gross pay.
TotalTax = .18 * GrossPay           ' Calculate total tax.
```

+ Operator

Description Used to sum two numbers.

Syntax *result* = *operand1* + *operand2*

Remarks Although you can also use the + operator to concatenate two character strings, you should use the **&** operator for concatenation to eliminate ambiguity and provide self-documenting code.

When you use the + operator, you may not be able to determine whether addition or string concatenation is to occur. If at least one operand is not a **Variant**, the following rules apply.

If	Then
Both expressions are numeric data types (**Integer**, **Long**, **Single**, **Double**, or **Currency**)	Add.
Both expressions are **Strings**	Concatenate.
One expression is a numeric data type and the other is a **Variant** (other than a **Null**)	Add.
One expression is a **String** and the other is a **Variant** (other than a **Null**)	Concatenate.
One expression is a **String** and the other is a **Variant** of **VarType** 8 (**String**)	Concatenate.
One expression is a **Variant** containing Empty	Return the remaining operand unchanged as *result*.
One expression is a numeric data type and the other is a **String**	A `Type mismatch` error occurs.

If either operand is a **Null** (**VarType** 1), *result* is a **Null**.

If both operands are **Variant** expressions, the **VarType** of the operands determines the behavior of the + operator in the following way.

If	Then
Both **Variant** expressions are of **VarType** 2–7 (numeric data types)	Add.
Both **Variant** expressions are of **VarType** 8 (**String**)	Concatenate.
One **Variant** expression is of **VarType** 2–7 (a numeric data type) and the other is of **VarType** 8 (**String**)	Add.

For simple arithmetic addition involving only operands of numeric data types, the data type of *result* is usually the same as that of the most precise operand. The order of precision, from least to most precise, is **Integer**, **Long**, **Single**, **Double**, **Currency**. The following are exceptions to this order:

- When a **Single** and a **Long** are added together, the data type of *result* is converted to a **Double**.
- When the data type of *result* is a **Variant** of **VarType** 3 (**Long**), **VarType** 4 (**Single**), or **VarType** 7 (Date) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 5 (**Double**).
- When the data type of *result* is a **Variant** of **VarType** 2 (**Integer**) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 3 (**Long**).

If one or both operands are **Null** expressions, *result* is a **Null**. If both operands are Empty, *result* is Empty. However, if only one operand is Empty, the other operand is returned unchanged as *result*.

See Also & Operator, * Operator, – Operator, / Operator, \ Operator, ^ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example determines total tax by using the + operator to add state and federal tax. After the calculation, `TotalTax` equals 1800.

```
GrossPay = 10000                          ' Assign gross pay.
StateTax = .06 * GrossPay                 ' Calculate state tax.
FederalTax = .12 * GrossPay               ' Calculate federal tax.
TotalTax = StateTax + FederalTax          ' Calculate total tax.
```

This example uses the + operator to concatenate three strings. If `String1` contains "Microsoft" and `String2` contains "Corporation," "Microsoft Corporation" is returned after the concatenation. Note that the space between the two words is the string literal `" "`.

```
= String1 + " " + String2
```

– Operator

Description Used to find the difference between two numbers or to indicate the negative value of an operand.

Syntax 1 *result = operand1 – operand2*

Syntax 2 *–number*

Remarks In Syntax 1, the – operator is the arithmetic subtraction operator used to find the difference between two numbers. The operands can be any numeric expression.

The data type of *result* is usually the same as that of the most precise operand. The order of precision, from least to most precise, is **Integer**, **Long**, **Single**, **Double**, **Currency**. When the operands are **Variant** expressions, the following rules supersede this order:

- When subtraction involves a **Single** and a **Long**, the data type of *result* is converted to a **Double**.
- When the data type of *result* is a **Variant** of **VarType** 3 (**Long**), **VarType** 4 (**Single**), or **VarType** 7 (Date) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 5 (**Double**).
- When the data type of *result* is a **Variant** of **VarType** 2 (**Integer**) that overflows its legal range, *result* is converted to a **Variant** of **VarType** 3 (**Long**).

If one or both operands are **Null** expressions, *result* is a **Null**. If an operand is Empty (**VarType** 0), it is treated as if it were 0.

In Syntax 2, the – operator is used as the unary negation operator to indicate the negative value of an operand. As with Syntax 1, the operand can be any numeric constant, variable, or expression or any function that returns a number.

See Also * Operator, + Operator, / Operator, \ Operator, ^ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example determines net pay by using the – operator to subtract the total of state and federal taxes from gross pay. After the calculation, `NetPay` equals 8200.

```
GrossPay = 10000                          ' Assign gross pay.
StateTax = .06 * GrossPay                 ' Calculate state tax.
FederalTax = .12 * GrossPay               ' Calculate federal tax.
TotalTax = StateTax + FederalTax          ' Calculate total tax.
NetPay = GrossPay - TotalTax              ' Calculate net pay.
```

This example uses the – operator to indicate a negative value.

```
-1200.55
```

/ Operator

Description Used to divide two floating-point numbers.

Syntax *result* = *operand1* / *operand2*

Remarks The operands can be any numeric expression.

The data type of *result* is usually a **Double** or a **Variant** of **VarType** 5 (**Double**). The following are exceptions to this rule:

- When both operands are **Integer** or **Single** expressions, *result* is a **Single** unless it overflows its legal range, in which case, *result* is a **Double**.
- When both operands are **Variant** expressions of **VarType** 2 (**Integer**) or **VarType** 4 (**Single**), *result* is a **Variant** of **VarType** 4 (**Single**) unless it overflows its legal range, in which case, *result* is a **Variant** of **VarType** 5 (**Double**).

If one or both operands are **Null** expressions, *result* is a **Null**. Any operand that is Empty (**VarType** 0) is treated as 0.

See Also * Operator, + Operator, – Operator, \ Operator, ^ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example divides 100,000 units among seven people and returns 14,285.7142857143.

```
= 100000 / 7
```

\ Operator

Description Used to divide two integers; returns an **Integer** or a **Long**.

Syntax *result* = *operand1* \ *operand2*

Remarks The operands can be any numeric expression.

Before division is performed, the operands are rounded to **Integer** or **Long** expressions.

Usually, the data type of *result* is an **Integer**, a **Long**, or a **Variant** of **VarType** 2 (**Integer**) or **VarType** 3 (**Long**) regardless of whether or not *result* is a whole number. Any fractional portion is truncated. However, if any operand is a **Null**, *result* is also a **Null**. Any operand that is Empty (**VarType** 0) is treated as 0.

See Also * Operator, + Operator, – Operator, / Operator, ^ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example divides 100,000 units among seven people using integer division and returns 14,285.

```
= 100000 \ 7
```

^ Operator

Description Used to raise a number to the power of an exponent.

Syntax *result* = *number* ^ *exponent*

Remarks The *number* and *exponent* operands can be any numeric expression. However, the *number* operand can be negative only if *exponent* is an integer value. When more than one exponentiation is performed in a single expression, the ^ operator is evaluated as it is encountered from left to right.

Usually, the data type of *result* is a **Double** or a **Variant** (**VarType** 5). However, if either *number* or *exponent* is a **Null** expression, *result* is also a **Null**.

See Also * Operator, + Operator, – Operator, / Operator, \ Operator, Comparison Operators, **Mod** Operator, Operator Precedence, **VarType** Function

Example This example returns 256 by raising 2 to the eighth power.

```
= 2 ^ 8
```

Abs Function

Description Returns the absolute value of a number.

Syntax **Abs**(*number*)

Remarks The argument *number* can be any valid numeric expression. The absolute value of a number is its unsigned magnitude. For example, `ABS(-1)` and `ABS(1)` both return 1.

The data type of the return value is the same as that of the *number* argument. However, if *number* is a **Variant** of **VarType** 8 (**String**) and can be converted to a number, the return value will be a **Variant** of **VarType** 5 (**Double**). If the numeric expression results in a **Null**, **Abs** returns a **Null**.

See Also **Atn** Function, **Cos** Function, Derived Math Functions, **Exp** Function, **Fix** Function, **Int** Function, **Log** Function, **Rnd** Function, **Sgn** Function, **Sin** Function, **Sqr** Function, **Tan** Function

Example This example uses **Abs** to determine the absolute difference between two numbers. The difference is 157.

```
X1 = -27: X2 = 130                    ' Identify two numbers.
Difference = Abs(X1 - X2)             ' Find range of numbers.
```

Access Basic Data Types

Description The following table shows the fundamental data types supported by Access Basic and the type-declaration suffix, storage size, and range of each data type.

Data type	Suffix	Storage size	Range
Integer	%	2 bytes	–32,768 to 32,767.
Long (long integer)	&	4 bytes	–2,147,483,648 to 2,147,483,647.
Single (single-precision floating-point)	!	4 bytes	–3.402823E38 to –1.401298E–45 for negative values; 1.401298E–45 to 3.402823E38 for positive values; and 0.
Double (double-precision floating-point)	#	8 bytes	–1.79769313486232E308 to –4.94065645841247E–324 for negative values; 4.94065645841247E–324 to 1.79769313486232E308 for positive values; and 0.
Currency (scaled integer)	@	8 bytes	–922,337,203,685,477.5808 to 922,337,203,685,477.5807.
String	$	1 byte per character	0 to approximately 65,535 bytes. (Some storage overhead is required.)

Data type	Suffix	Storage size	Range
Variant	(None)	As appropriate	Any numeric value up to the range of a **Double** or any character text.
User-defined (using **Type**)	(None)	Number required by elements	The range of each element is the same as the range of its fundamental data type, listed above.

See Also **Currency** Data Type, **Double** Data Type, **Integer** Data Type, **Long** Data Type, **Single** Data Type, **String** Data Type, **Type** Statement, **Variant** Data Type

ActiveControl Property

Applies To **Screen** object.

Description Used to refer to the control that has the focus.

Setting This property has no settings. You can use it to refer to an active control together with one of its properties or methods or to the value of the control. For example, you can assign the value of the control to a variable using this syntax:

```
Set X = Screen.ActiveControl
```

Usage Design view: read-only. Other views: read-only.

Remarks Use the ActiveControl property to refer to the control that has the focus at run time. The following example returns the active control's ControlName property:

```
X = Screen.ActiveControl.ControlName
```

If no form has the focus when you use the ActiveControl property, or if all of the active form's controls are hidden or disabled, an error occurs.

See Also ActiveForm Property, ActiveReport Property

ActiveForm Property

Applies To **Screen** object.

Description Used to refer to the form that has the focus.

Setting This property has no settings. You can use it to refer to an active form together with one of its properties or methods, or to the value of the form. For example, you can assign the value of the form to a variable using this syntax:

```
Set X = Screen.ActiveForm
```

Usage Design view: read-only. Other views: read-only.

Remarks Use the ActiveForm property to refer to the form that has the focus at run time. The following example returns the active form's FormName property:

```
Dim MyForm As Form
Set MyForm = Screen.ActiveForm
MsgBox "Current form is " & MyForm.FormName
```

If a subform has the focus, ActiveForm refers to the main form. If no form or subform has the focus when you use the ActiveForm property, an error occurs.

See Also ActiveControl Property, ActiveReport Property

ActiveReport Property

Applies To **Screen** object.

Description Used to refer to the report that has the focus.

Setting This property has no settings. You can use it to refer to an active report together with one of its properties or methods, or to the value of the report. For example, you can assign the value of the report to a variable using this syntax:

```
Set X = Screen.ActiveReport
```

Usage Design view: read-only. Other views: read-only.

Remarks Use the ActiveReport property to refer to the report that has the focus at run time. The following example returns the active report's FormName property:

```
Dim MyReport As Report
Set MyReport = Screen.ActiveReport
MsgBox "Current report is " & MyReport.FormName
```

If no report has the focus when you use the ActiveReport property, an error occurs.

See Also ActiveControl Property, ActiveForm Property

AddColon, AutoLabel Properties

Apply To Tools (bound object frame, check box, combo box, command button, list box, option button, option group, subform/subreport, text box, toggle button).

Description

- AddColon—determines whether a colon follows the text in labels for new controls.
- AutoLabel—determines whether labels are attached to new controls.

Setting To see the property sheet, choose the Properties button on the tool bar or the Properties command from the View menu. Select the tool in the toolbox. The default property settings depend on the specific tool you select.

The AddColon property settings are:

Setting	Description
Yes	A colon follows the text in labels for new controls.
No	A colon does not follow the text in labels for new controls.

The AutoLabel property settings are:

Setting	Description
Yes	A label is attached to new controls.
No	A label is not attached to new controls.

Remarks Most controls you create have an attached label with a colon following the label text. Changes to this default control property setting affect only controls created on the current form or report. To change the default control settings for all new forms or reports, create a new template.

Access Basic These properties aren't available from Access Basic.

AddMenu Action

Description

Adds a drop-down menu to a custom menu bar for a form.

To create a custom menu bar, you must do the following:

- Create a menu bar macro that contains an AddMenu action for each drop-down menu you want on the custom menu bar.
- Assign commands to each drop-down menu by creating a macro group for each menu. Each command runs the set of actions defined by a macro in this macro group.
- Attach the menu bar macro to the form by entering the name of the menu bar macro in the form's OnMenu property.

Action argument	Description
Menu Name	The name of the drop-down menu to add to the custom menu bar. Required argument.
Menu Macro Name	The name of the macro group that contains the macros for the menu's commands. Required argument.
Status Bar Text	The text to display in the status bar when the menu is selected.

Remarks

Each drop-down menu on the menu bar requires a separate AddMenu action.

The custom menu bar replaces the Microsoft Access menu bar for the form. If you want to retain certain Microsoft Access commands to use on menus on the custom menu bar, use the DoMenuItem action to put the commands into the macro groups for the desired menus.

You can run a macro from a menu command by using the RunMacro action in the macro for the command.

Note AddMenu actions can be used only in a menu bar macro attached to the OnMenu property of a form, and menu bar macros can contain only AddMenu actions.

Access Basic

This action isn't available from Access Basic.

AddNew Method

Description Prepares a new record in a specified **Table** or **Dynaset**.

Syntax *object*.**AddNew**

Remarks The **AddNew** method uses the following argument.

Argument	Description
object	Name of an open **Table** or **Dynaset** object to which a new record will be added

The **AddNew** method prepares a new record for insertion at the end of the table or **Dynaset** named by *object*. It sets the fields to **Null** (the default values specified in the **Table's** Design view aren't used).

You don't have to use the **Edit** method to edit the new record (you still must use **Edit** before you can edit an existing record, however). After you edit the new record, use the **Update** method to save the changes and add the record to *object*.

When you use **AddNew** and Microsoft Access has to create a new page to hold the new record, page locking is pessimistic. If the new record will fit in an existing page, page locking is optimistic.

If you add a record to a **Table**, the record won't appear in any **Dynaset** based on that **Table**. If you add a record to a **Dynaset**, however, that record will appear in the underlying **Table**.

The new record doesn't become the current record. The record that was current before you used **AddNew** remains current. If you want to make the new record current, you can use the **MoveLast** method (if *object* refers to a **Dynaset** or a nonindexed **Table**) or the **Seek** method (if *object* refers to an indexed **Table**) immediately after you save the new record with **Update**.

If *object* refers to an indexed **Table**, the new record appears in sequence according to the index, even though it was actually added to the end of the table.

If *object* refers to a table that isn't open when you use **AddNew**, an error occurs.

See Also **Delete** Method; **Edit** Method; **MoveFirst, MoveLast, MoveNext, MovePrevious** Methods; **Seek** Method; **Update** Method

Example This example prepares a new record in the Customers table of the NWIND.MDB database, sets new values, and saves the changes.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Customers")          ' Open Table.
MyTable.Index = PrimaryKey
MyTable.AddNew                                     ' Prepare new record.
MyTable("Customer ID") = "FINNF"                   ' Set record key.
MyTable("Company Name") = "Finnegan's Foods"       ' Set company name.
MyTable.Update                                     ' Save changes.
MyTable.Seek "=", "FINNF"                          ' Make new record current.
MyTable.Delete                                     ' Delete record.
MyTable.Close                                      ' Close Table.
```

AfterUpdate Property

See BeforeUpdate, AfterUpdate Properties

AllowEditing, DefaultEditing Properties

Apply To Forms.

Description

- AllowEditing—determines whether the Allow Editing command on the Records menu in Form view is enabled when you open a form from the Database window.
- DefaultEditing—determines whether a form opens to allow editing, read-only access, or data entry.

Setting The AllowEditing property settings are:

Setting	Description
Available	(Default) The Allow Editing command is enabled.
Unavailable	The Allow Editing command is disabled.

The DefaultEditing property settings are:

Setting	Description
Allow Edits	(Default) You can edit, add, or delete records in the form's table or query.
Read Only	You can't edit, add, or delete records until you choose the Allow Editing command.
Data Entry	You can add new records. Existing records aren't available. (See note below.)

Remarks

Use these properties to design forms for data entry or for read-only access. You can combine them to establish the following conditions.

AllowEditing	DefaultEditing	Description
Available	Allow Edits	All records are available. You can edit, add, or delete records.
Available	Read Only	All records are available. You must choose the Allow Editing command to edit, add, or delete records.
Available	Data Entry	No existing records are available. You can add new records.
Unavailable	Allow Edits	All records are available. You can edit, add, or delete records.
Unavailable	Read Only	All records are available. You can't edit, add, or delete records.
Unavailable	Data Entry	No existing records are available. You can't edit, add, or delete records. (This combination is not recommended.)

Note The Data Entry setting doesn't completely prevent you from viewing records using the form. If the AllowFilters property is set to Yes, you can choose the Show All Records command from the Records menu to display all records.

Access Basic

The AllowEditing property settings and their values are:

Setting	Value
Available	–1
Unavailable	0

Design view: read/write. Other views: read-only.

The DefaultEditing property settings and their values are:

Setting	Value
Data Entry	1
Allow Edits	2
Read Only	3

Design view: read/write. Other views: read-only.

AllowFilters Property

Applies To Forms.

Description Determines whether records can be filtered.

Setting The AllowFilters property settings are:

Setting	Description
Yes	(Default) Records can be filtered.
No	Records can't be filtered.

Remarks When you use a filter, you apply criteria to display only records that meet specific conditions. In an Employees table, for example, you could show only records of employees with over five years of service. You could also use a filter to restrict access to records containing sensitive information, such as financial or medical data.

When the AllowFilters property is set to No, the following commands on the Records menu are disabled: Edit Filter/Sort, Apply Filter/Sort, and Show All Records.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also DefaultView, ViewsAllowed Properties

AllowUpdating Property

Applies to Forms.

Description Determines which tables you can edit when a form is in Form view or Datasheet view.

Setting The AllowUpdating property settings are:

Setting	Description
Default Tables	(Default) Only the default tables and controls bound to their fields can be edited.
Any Tables	All tables and controls bound to their fields can be edited.
No Tables	No tables or controls bound to their fields can be edited.

Remarks You can create forms based on multiple underlying tables, whose fields are bound to controls on the forms. Depending on the AllowUpdating property setting, you can limit which of these bound controls can be edited.

The AllowUpdating property prevents editing of all controls that are bound to the tables set by AllowUpdating. For example, if you set this property to No Tables, you won't be able to edit any of the bound controls. You can view the data, but you can't change it using the form.

Note In addition to the editing control provided by AllowUpdating, each control on a form has a Locked property that you can set to determine whether the control and its underlying data can be edited. If the Locked property is set to Yes, you can't edit the data.

Access Basic The property settings and their values are:

Setting	Value
Default Tables	0
Any Tables	1
No Tables	2

Design view: read/write. Other views: read-only.

And Operator

Description Used to perform a logical conjunction on two expressions.

Syntax *result* = *expr1* **And** *expr2*

Remarks If and only if both expressions evaluate true (nonzero), *result* is **True** (–1). If either expression evaluates false (0), *result* is **False** (0). The following table illustrates how *result* is determined.

If *expr1* is	And *expr2* is	*result* is
true (nonzero)	true	**True** (–1)
true	false (0)	**False** (0)
true	**Null**	**Null**
false	true	**False**
false	false	**False**
false	**Null**	**False**
Null	true	**Null**
Null	false	**False**
Null	**Null**	**Null**

The **And** operator also performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

If bit in *expr1* is	And bit in *expr2* is	*result* is
0	0	0
0	1	0
1	0	0
1	1	1

Bit-wise comparisons can be performed only in Access Basic code.

See Also **Eqv** Operator, **Imp** Operator, **Not** Operator, Operator Precedence, **Or** Operator, **VarType** Function, **Xor** Operator

Example This example prints a message that depends on the value of variables `A`, `B`, and `C`, assuming that no variable is a **Null**. If `A = 10`, `B = 8`, and `C = 6`, both expressions evaluate **True**. Because both expressions are **True**, the **And** expression is also **True**.

```
If A > B And B > C Then
  Debug.Print "Both expressions are True."
Else
  Debug.Print "One or both expressions are False."
End If
```

Any Data Type

Description The **Any** data type is used only to suppress type checking of arguments passed to dynamic-link library (DLL) procedures.

AppActivate Statement

Description Activates an application window.

Syntax **AppActivate** *titletext*

Remarks The argument *titletext* is a string expression that is the name that appears in the title bar of the application window to activate. If there is more than one instance of the application, the operating environment arbitrarily selects the one to activate. The name that appears in the title bar and *titletext* must match character for character, but the comparison is not case-sensitive: for example, the words Calculator and calculator would match.

The **AppActivate** statement moves the focus to the named application window but doesn't affect whether the window is maximized or minimized. The focus leaves the activated application window when the user takes some action to move the focus or close the window.

See Also **SendKeys** Statement, **Shell** Function

Example

This example shells out of the active application to the Calculator application included with Microsoft Windows™ and then uses the **SendKeys** statement to send keystrokes to either quit the Calculator application or add some numbers, depending on the user's choice. **AppActivate** is used to move the focus between Microsoft Access and the Calculator.

```
AppActivate "Microsoft Access - Jon King"  ' Change focus to Microsoft
                                           ' Access window.
SendKeys "%{ }{Down 3}{Enter}", True       ' Minimize Microsoft Access.
X = Shell("CALC.EXE", 1)                   ' Shell the Calculator.
For I = 1 To 100                           ' Set up counting loop.
  SendKeys I & "{+}", True                 ' Send keystrokes to Calculator
                                           '  to add each value of I.
Next I                                     ' Get grand total.
SendKeys "=", True                           ' Move focus back.
AppActivate "Microsoft Access - Jon King"    ' Stop to see results.
Msg = "Choose OK to close the Calculator."
MsgBox Msg
AppActivate "Calculator"                   ' Move focus back to Calculator.
SendKeys "%{F4}", True                     ' Alt+F4 to close Calculator.
AppActivate "Microsoft Access - Jon King"    ' Move focus back.
SendKeys "%{ }{Enter}", True               ' Restore Microsoft Access to
                                           ' size.
```

AppendChunk Method

Description

Appends data from a string to a Memo or OLE Object field in the copy buffer of a specified **Table** or **Dynaset**.

Syntax

object ! *field*.**AppendChunk**(*source*)

Remarks

The **AppendChunk** method uses the following arguments.

Argument	Description
object	Name of a **Table** or **Dynaset** object
field	Name of a Memo or OLE Object field
source	String expression that is the name of an object containing the data you want to append to *field*

You can use the **AppendChunk** and **GetChunk** methods to create an OLE Object field that exceeds the Access Basic 64K string size limitation.

You can also use these methods to conserve string space when working with Memo fields, which are limited to 32K bytes. Certain operations (copying, for example) involve temporary strings. If string space is limited, you may need to work with chunks of a memo instead of the entire field.

If you want to clear the current contents of *field*, set the field to a zero-length string. The following example shows how you can clear a field named `Notes`.

```
MyTable!Notes = ""
```

If there is no current record when you use **AppendChunk**, an error occurs.

See Also **FieldSize** Method, **GetChunk** Method

Example This example uses the **GetChunk** method to return consecutive 16K chunks of the Notes field of the Employees table in the NWIND.MDB database. It copies the chunks to the string array `NoteArray( )`, where they can be changed. **AppendChunk** then copies `NoteArray( )` back to `Notes`.

```
Option Base 1 ' This line should appear in the Declarations section.

Sub Get_Notes
  Const ChunkSize = 16384                          ' Set size of chunk.
  Dim NumChunks As Integer, TotalSize As Long, X As Integer
  Dim MyDB As Database, MyTable As Table
  Set MyDB = CurrentDB()
  Set MyTable = MyDB.OpenTable("Employees")        ' Open Table.
  TotalSize = MyTable!Notes.FieldSize()            ' Get field size.
                                                   ' How many chunks?
  NumChunks = TotalSize \ ChunkSize - (TotalSize Mod ChunkSize <> 0)
  ReDim NoteArray(NumChunks) As String * ChunkSize
                                                   ' Get current Notes field.
  For X = 1 To NumChunks
    NoteArray(X) = MyTable!Notes.GetChunk((X - 1) * ChunkSize, ChunkSize)
  Next X
  ...                                              ' Make changes.
  MyTable.Edit                                     ' Enable editing.
    MyTable!Notes = ""                             ' Initialize Notes field.
  For X = 1 To NumChunks
    MyTable!Notes.AppendChunk(NoteArray(X))        ' Replace with edited Notes.
  Next X
  MyTable.Update                                   ' Save changes.
  MyTable.Close                                    ' Close Table.
End Sub
```

ApplyFilter Action

Description

Applies a filter, a query, or an SQL WHERE clause to a form or report to restrict or sort the records from the form's or report's underlying table or query. For reports, you can use this action only in a macro attached to the report's OnOpen property.

Action argument	Description
Filter Name	The name of a filter or query that restricts or sorts the form's or report's records. You can enter the name of either an existing query or a filter that has been saved as a query.
Where Condition	A valid SQL WHERE clause or expression that restricts the form's or report's records. In contrast to other action arguments, if you use only the name of a control in the expression for this argument, it refers to a control on the form or report that you are applying the filter to, not the form or report that called the macro (although these two are frequently the same).

Note You must use one or both arguments. Use the Filter Name argument if you've already defined a filter that provides the appropriate data. Use the Where Condition argument to enter the restriction criteria directly. If you use both arguments, Microsoft Access applies the WHERE clause to the filter.

Remarks

You can apply a filter or query to a form in Form view or Datasheet view.

For forms, this action is similar to choosing the Apply Filter/Sort command from the Records menu or clicking the Apply Filter/Sort button on the tool bar. The command or button applies the most recently created filter to the form, whereas the action applies a specified filter or query.

If you choose the Edit Filter/Sort command from the Records menu or click the Edit Filter/Sort button on the tool bar after running the ApplyFilter action, the Filter window shows the filter criteria you have selected with this action.

To remove a filter and display all of the form's records, you can use the ShowAllRecords action, the Show All Records command on the Records menu, or the Show All Records button on the tool bar.

See Also

OpenQuery Action, ShowAllRecords Action

Access Basic

Syntax

DoCmd ApplyFilter [*filtername*] [, *wherecondition*]

Argument	Description
filtername	A string expression that is the valid name of a query in the current database
wherecondition	A string expression that is a valid SQL WHERE clause

Remarks

You must include at least one of the two ApplyFilter arguments. If you enter a value for both arguments, *wherecondition* is applied to the filter.

If you specify the *wherecondition* argument and leave the *filtername* argument blank, you must include the *filtername* argument's comma. If you leave both arguments blank, don't use a comma.

Example

This example uses the ApplyFilter action to display only records that contain the name `King` in the `Last Name` field.

```
DoCmd ApplyFilter , "[Last Name] = ""King"""
```

Asc Function

Description Returns a numeric value that is the ANSI code for the first character in a string expression.

Syntax **Asc**(*stringexpression*)

Remarks If the *stringexpression* argument is a zero-length **String** or a **Null Variant** (**VarType** 1), a run-time error will occur.

See Also Appendix A, "ANSI Character Set"; **Chr**, **Chr$** Functions

Example This example returns 65, the ANSI character code for an uppercase letter A.

```
= Asc("A")
```

Atn Function

Description Returns the arctangent of a number.

Syntax **Atn**(*number*)

Remarks The argument *number* can be any valid numeric expression.

Atn takes the ratio (*number*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

The result is expressed in radians and is in the range –Pi/2 to Pi/2 radians. Pi approximately equals 3.141593.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Note **Atn** is the inverse trigonometric function of **Tan**, which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse **Atn** with the cotangent, which is the simple inverse of a tangent (1/tangent).

See Also **Cos** Function, **Sin** Function, **Tan** Function

Example This example uses **Atn** to calculate Pi. By definition, `Atn(1)` is 45 degrees; 180 degrees equals Pi radians.

```
= 4 * Atn(1)
```

AutoLabel Property

See AddColon, AutoLabel Properties

AutoRepeat Property

Applies To

Control (command button) on a form.

Description

Determines whether a macro runs repeatedly while a command button is pressed.

Setting

The AutoRepeat property settings are:

Setting	Description
Yes	The macro named in the OnPush property runs repeatedly while the command button is pressed.
No	(Default) The macro runs once.

Remarks

When you choose a command button, Microsoft Access runs the macro specified in the button's OnPush property. Use the AutoRepeat property to have the macro run repeatedly for as long as the command button is held down. For example, if the macro displays a form's next record, it can continue cycling through records until the command button is released.

The initial repeat of the macro occurs 0.5 second after it first runs. Subsequent repeats occur either 0.25 second apart or the duration of the macro, whichever is longer.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

AutoResize Property

Applies To Forms.

Description Determines whether a Form window opens automatically sized to display complete records.

Setting The AutoResize property settings are:

Setting	Description
Yes	The Form window displays multiple complete records when opened in Form view. To display single complete records, set the DefaultView property to Single Form. If the RecordSource property is blank, the form displays a single record.
No	When opened, the Form window has the same size as when last closed.

Remarks The Form window resizes only if opened in Form view. If you open the form first in Design view or Datasheet view and then change to Form view, the Form window doesn't resize.

A Form window opens with its upper-left corner in the same location as when it was closed.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also MoveSize Action, RecordSource Property

Avg Function

Description Calculates the arithmetic mean of a set of values contained in a specified field in a query, form, or report.

Syntax **Avg**(*expr*)

Remarks The **Avg** function uses the following argument.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to average, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

The average calculated by **Avg** is the arithmetic mean (the sum of the values divided by the number of values). You might use **Avg**, for example, to calculate average freight cost.

The **Avg** function doesn't include any **Null** fields in the calculation.

You can use **Avg** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **DAvg** Function; **Count** Function; **First**, **Last** Functions; **Min**, **Max** Functions; **StDev**, **StDevP** Functions; **Sum** Function; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to calculate the average sale for orders over $1000. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT Avg([Order Amount]) FROM Orders WHERE [Order Amount] > 1000;
```

The next example calculates the average sale for all of the underlying records in a form that also uses the Orders table in the NWIND.MDB. To apply a condition that limits the calculation to only some records, such as those for orders greater than $1000, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= Avg([Order Amount])
```

BackColor Property

Applies To Form sections. Report sections. Tools and controls (combo box, graph, label, list box, option group, rectangle, text box, unbound object frame).

Description Specifies the color for the interior of a control.

Setting Use the Palette to set this property in Design view.

Remarks If available, the BorderStyle and BackStyle properties must be set to Normal.

Use this property with Access Basic to create special visual effects on custom reports when you print with a color printer.

Access Basic The value of this property is a **Long** data type.

Design view: read/write. Other views: read-only.

See Also DrawMode Property, DrawStyle Property, DrawWidth Property, FillColor Property, OnFormat Property, OnPrint Property, **PSet** Method

BackStyle Property

Applies To Tools and controls (graph, label, option group, rectangle, unbound object frame).

Description Determines whether a control is transparent.

Setting Set this property using the property sheet or the Palette.

The BackStyle property settings are:

Setting	Description
Normal	(Default for all controls except option group) The control has the color set by the BackColor property.
Clear	(Default for option group) The control is transparent and has the color of the form or report behind it.

Remarks To use the BackStyle property, the SpecialEffect property must be set to Normal.

Access Basic

The property settings and their values are:

Setting	Value
Normal	1
Clear	0

Design view: read/write. Other views: read-only.

See Also BorderColor Property, BorderStyle Property, BorderWidth Property, FillStyle Property

Beep Action

Description Sounds a beep tone through the computer's speaker.

Remarks You can use the Beep action to signal the following occurrences:

- Important screen changes have occurred.
- The wrong kind of data has been entered in a field. For example, the user has entered numeric data in a Text field.
- A macro has reached a certain point or has completed its actions.

See Also MsgBox Action; ValidationRule, ValidationText Properties

Access Basic

Syntax

DoCmd Beep

Remarks

This action takes no arguments.

Beep Statement

Description Sounds a tone through the computer speaker.

Syntax **Beep**

Remarks The frequency and duration of the beep depend on hardware, which may vary between computers.

See Also **Beep** Action

Example This example uses **Beep** to sound a tone in the computer speaker if `Answer` is less than 1 or greater than 3.

```
Do
Answer = InputBox("Enter a value from 1 to 3.")
If Answer >= 1 And Answer <= 3 Then       ' Check range.
  Msg = "You entered a value in the proper range."
  Exit Do                                 ' Exit Do...Loop.
Else
  Beep                                    ' Beep if not in range.
End If
Loop
MsgBox Msg                                ' Display results.
```

BeforeUpdate, AfterUpdate Properties

Apply To Forms. Controls (check box,* combo box, list box, option button,* option group, text box, toggle button*) on a form.

*Except when the control is in an option group.

Description Specify the name of a macro or user-defined function to run before or after one of the following events:

- BeforeUpdate—before changed data in a control or record is updated.
- AfterUpdate—after changed data in a control or record is updated.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname( )*.

Remarks Within a record, changed data in each control is updated when the control loses the focus (or when the user presses ENTER or TAB). Microsoft Access runs the macro or function specified by the BeforeUpdate or AfterUpdate property for the control before or after the data is updated. If the data in the control is not changed, neither macro or function runs.

When the focus leaves the record, or if the user chooses the Save Record command from the File menu, the entire record is updated, and the data is saved in the database. Microsoft Access runs the macro or function specified by the BeforeUpdate or AfterUpdate property for the form before or after the record is updated. If the data in the record is not changed, neither macro or function runs.

You often use the BeforeUpdate property and a macro to validate data, especially when you perform complex validations, such as:

- A validation that involves more than one control on a form.
- A validation that checks to see if a control has a value.
- A validation for which you want to display different error messages for different data entered.
- A validation that the user can override.
- A validation that involves complex calculations or more than one condition.

Note If you can validate data using a single expression, use the ValidationRule property.

You can also use BeforeUpdate macros if you want to store all the validations for a form in one macro group, or if you use a validation expression on several forms.

For example, suppose you have a form on which the user enters postal codes for different countries in a Postal Code field. You use conditional expressions in the macro to check the value of the Country field on the form and then display the appropriate error message for the value entered in the Postal Code field.

You can use the AfterUpdate property to display a different page in the form or move the focus to a particular control or record. For example, when the user enters a value in a Category control on a form, you can use the GoToPage action in an AfterUpdate macro to move the focus to the page of the form that contains the fields for this category.

You can use the CancelEvent action in a BeforeUpdate macro to cancel the update. If the BeforeUpdate macro is attached to a control, the CancelEvent action returns the focus to the control, and the data the user entered remains in the control. You can restore the previous value of the control by using the DoMenuItem action in the BeforeUpdate macro to run the Undo Current Field command, or the SendKeys action to send an ESC key (which has the same effect as running the Undo Current Field command).

If the BeforeUpdate macro is attached to a form (in which it runs for each changed record), the CancelEvent action returns the focus to the record. You can use the DoMenuItem action in the BeforeUpdate macro to run the Undo Current Record command (or the SendKeys action to send an ESC key) to restore the record to its previous value.

You can't use the CancelEvent action in an AfterUpdate macro.

Access Basic

Use a string expression to set the value of each property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign (=) and followed by parentheses.

Design view: read/write. Other views: read-only.

See Also

OnEnter, OnExit Properties

BeginTrans Statement

Description Starts a transaction.

Syntax **BeginTrans**

Remarks A transaction is a series of changes you make to a recordset. You mark the beginning of a transaction with the **BeginTrans** statement. You use the **Rollback** or **CommitTrans** statement to end a transaction.

You can use **Rollback** to undo and **CommitTrans** to save changes made during the current transaction. After you make changes to your database, you can decide whether to undo the changes or commit (save) them. If you nest a transaction, you must roll back or save the current transaction before you can roll back or save one at an earlier level.

You can nest up to five levels of transactions by using multiple **BeginTrans** statements. Once you start a transaction, use additional **BeginTrans** statements to isolate smaller groups of changes. You can then roll back or save changes to these smaller groups instead of to an entire larger transaction.

Caution If a recordset is based entirely on Microsoft Access tables, its Transactions property is **True** (–1) and you can use transactions. Tables created by other database products, however, may not support transactions. For example, you cannot use transactions in a recordset based on a Paradox table. In this case, the Transactions property is **False** (0). You should inspect the Transactions property before using the **BeginTrans** statement to make sure that the recordset supports transactions. When used with a nonsupported recordset, **BeginTrans** is ignored; no error occurs.

Note Transactions in Access Basic are independent of the transactions performed in Microsoft Access forms and reports.

See Also **CommitTrans** Statement, **Rollback** Statement, Transactions Property, **Update** Method

Example This example changes the job title of all sales representatives in the Employees table of the NWIND.MDB database. It uses **BeginTrans** to start a transaction that isolates all of the changes made to the Employees table. Notice that **Rollback** can be used to undo changes saved with the **Update** method.

```
Const MB_ICONQUESTION = 32
Const YES = 6
Const YES_NO = 4
Dim CRLF As String, EmployeeName As String
Dim Message As String, Prompt As String
Dim MyDB As Database, MyTable As Table
```

```
CRLF = Chr$(13) + Chr$(10)
Prompt = "Change title to Account Executive?"
Set MyDB = CurrentDB()                                  ' Get current database.
Set MyTable = MyDB.OpenTable("Employees")               ' Open Table.
BeginTrans                                              ' Begin transaction.
Do Until MyTable.EOF
  If MyTable!Title = "Sales Representative" Then
    EmployeeName = MyTable("Last Name") + ", " + MyTable("First Name")
    Message = "Employee: " + EmployeeName + CRLF + CRLF
    If MsgBox(Message + Prompt, MB_ICONQUESTION + YES_NO, "Change Job
    ➥ Title") = YES Then
      MyTable.Edit                                      ' Enable editing.
      MyTable!Title = "Account Executive"               ' Change title.
      MyTable.Update                                    ' Save changes.
    End If
  End If
  MyTable.MoveNext                                      ' Move to next record.
Loop
If MsgBox("Save all changes?", MB_ICONQUESTION + YES_NO, "Save Changes")
➥ = YES Then
  CommitTrans                                           ' Commit changes.
Else
  Rollback                                              ' Undo changes.
End If
MyTable.Close                                           ' Close Table.
```

Tip Using an update query to change job titles might be more efficient.

Between...And Operator

Description Determines whether the value of an expression lies within a specified range of values.

Syntax *expr* [**Not**] **Between** *value1* **And** *value2*

Remarks The **Between...And** operator uses the following arguments.

Argument	Description
expr	Expression identifying the field that contains the data you want to evaluate
value1, value2	Expressions against which you want to evaluate *expr*

If the value of *expr* is between *value1* and *value2* (inclusive), the **Between...And** operator returns **True** (–1); otherwise, it returns **False** (0). You can include the **Not** logical operator to evaluate the opposite condition (that is, whether *expr* lies outside the range defined by *value1* and *value2*).

You might use **Between...And** to determine whether the value of a control falls within a specified numeric range. The following example determines whether an order was shipped to a location within a range of postal codes. If the `Ship Postal Code` is between `98101` and `98199`, the **IIf** function returns `"Local"`. Otherwise, it returns `"Nonlocal"`.

```
=IIf([Ship Postal Code] Between 98101 And 98199, "Local", "Nonlocal")
```

If *expr*, *value1*, or *value2* is **Null**, **Between...And** returns a **Null**.

You can use the **Between...And** operator in a query expression or in a calculated control on a form or report.

See Also **In** Operator, **Like** Operator, **Not** Operator

Example The following example uses the Orders table in the NWIND.MDB database to create a query that includes orders shipped to the local area. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT * FROM Orders WHERE [Ship Postal Code] Between 98101 And 98199;
```

The next example sets the text for a control on a form based on the Orders table in the NWIND.MDB database. Depending on where an order is shipped, it sets the text for the control. You can enter this expression in a calculated control on the form.

```
=IIf([Ship Postal Code] Between 98101 And 98199, "Next-Day Delivery",
➥ "2-Day Delivery")
```

BOF Property

Applies To Recordsets.

Description Indicates whether the current record position is before the first record in a recordset (at the beginning of a file).

Setting The BOF property settings are:

Setting	Description
–1	The current record position is before the first record in a recordset.
0	The current record position is on or after the first record in a recordset.

Usage Read-only.

Remarks You can use the BOF property to determine whether a recordset contains records or whether you have gone beyond the limits of a recordset as you move from record to record.

If the recordset contains no records, BOF is **True** (–1). It remains **True** until you move to an existing record. When you create or open a recordset that contains at least one record, the first record is the current record and BOF is **False** (0).

If the first record is the current record when you use **MovePrevious**, BOF is set to **True**. If you use **MovePrevious** while BOF is **True**, an error occurs; BOF remains **True** and there is no current record.

See Also EOF Property, NoMatch Property

Example This example uses the BOF and EOF properties to detect the limits of the Orders table in the NWIND.MDB database. It moves through the records first from the beginning of the file to the end and then from the end to the beginning. Notice that there is no current record immediately following the first loop.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Orders")      ' Open Table.
Do Until MyTable.EOF                        ' Until end of file.
  MyTable.MoveNext                          ' Move to next record.
  ...
Loop
MyTable.MoveLast                            ' Move to last record.
Do Until MyTable.BOF                        ' Until beginning of file.
  MyTable.MovePrevious                      ' Move to previous record.
  ...
Loop
MyTable.Close                               ' Close Table.
```

Bookmark Property

Applies To Forms. Recordsets.

Description Sets a bookmark that you can later use to make a particular record current.

Setting The Bookmark property setting contains a randomly created string that uniquely identifies each record in a recordset.

Usage Read/write.

Remarks In Access Basic, you set a bookmark on the current record by assigning the value of the Bookmark property to a string variable. Then you can move to another record and later quickly return to the first record by setting the Bookmark property to that string variable.

There is no limit to the number of bookmarks you can set. To place a bookmark on a record other than the current record, move to the desired record and assign the value of the Bookmark property to a string variable that identifies this record.

For a form or recordset based entirely on Microsoft Access tables, the value of the Bookmarkable property is **True** (–1) and bookmarks can be used. Other database products may not support bookmarks, however. For example, you cannot use bookmarks in a form or recordset based on an attached table that has no primary index.

Inspect the value of the Bookmarkable property before you use the Bookmark property to make sure that the form or recordset supports bookmarks. If the form or recordset doesn't support bookmarks (that is, the Bookmarkable property is **False** (0)) and you use the Bookmark property, an error occurs.

You can also set a bookmark for the **Dynaset** used by a form. This enables you to mark which record is displayed by the form and then move to another record by setting the form's Bookmark property.

See Also Bookmarkable Property

Example This example sets a bookmark on a record in a **Dynaset** (based on the Orders table of the NWIND.MDB database) and uses the Bookmark property to return to that record.

```
Dim MyDB As Database, MySet As Dynaset, MyMark As String
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Orders")          ' Create Dynaset.
MySet.FindFirst "[Ship Country] = 'UK'"           ' Locate first UK order.
If MySet.Bookmarkable And Not MySet.NoMatch Then
  MyMark = MySet.Bookmark                         ' Set bookmark.
  MySet.FindFirst "[Ship Country] = 'USA'"        ' Locate USA order.
  ...
  MySet.Bookmark = MyMark                         ' Return to UK.
End If
```

Bookmarkable Property

Applies To Recordsets.

Description Indicates whether a recordset supports bookmarks, which can be used with the Bookmark property.

Setting The Bookmarkable property settings are:

Setting	Description
–1	The recordset supports bookmarks.
0	The recordset doesn't support bookmarks.

Usage Read-only.

Remarks For a recordset based entirely on Microsoft Access tables, the value of the Bookmarkable property is **True** (–1) and bookmarks can be used. Other database products may not support bookmarks, however. For example, you cannot use bookmarks in a recordset based on an attached table that has no primary index.

Inspect the value of the Bookmarkable property before you use the Bookmark property to make sure that the recordset supports bookmarks.

See Also Bookmark Property

Example This example determines whether a bookmark can be used in a **Dynaset** based on the Orders table in the NWIND.MDB database.

```
Dim MyDB As Database, MySet As Dynaset, MyMark As String
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Orders")        ' Create Dynaset.
MySet.FindFirst "[Ship Country] = 'UK'"         ' Locate first UK order.
If MySet.Bookmarkable And Not MySet.NoMatch Then
  MyMark = MySet.Bookmark                       ' Set bookmark.
  MySet.FindFirst "[Ship Country] = 'USA'"      ' Locate USA order.
  ...
  MySet.Bookmark = MyMark                       ' Return to UK.
End If
```

BorderColor Property

Apply To Tools and controls (bound object frame, combo box, graph, label, line, option group, rectangle, subform/subreport, text box, unbound object frame).

Description Specifies the color for the border of a control.

Setting Set this property using the Palette.

Remarks To use the BorderColor property, the SpecialEffect property must be set to Normal. If available, the BorderStyle and BackStyle properties must be set to Normal.

Access Basic Use a numeric expression to set the value of this property. You can use the **RGB** function or the **QBColor** function in the expression.

Design view: read/write. Other views: read-only.

See Also BorderWidth Property, ForeColor Property

BorderStyle Property

Applies To Tools and controls (bound object frame, combo box, graph, label, line, option group, rectangle, subform/subreport, text box, unbound object frame).

Description Determines whether a control's border is transparent or visible.

Setting Set this property using the property sheet or the Palette.

The BorderStyle property settings are:

Setting	Description
Clear	(Default for label, graph, object frame, subreport, and text box) The border is transparent.
Normal	(Default for all other controls) The border is visible.

Remarks To use the BorderStyle property, the SpecialEffect property must be set to Normal.

Access Basic

The property settings and their values are:

Setting	Value
Clear	0
Normal	1

Design view: read/write. Other views: read-only.

See Also

BorderColor Property, BackStyle Property

BorderWidth Property

Applies To

Tools and controls (bound object frame, combo box, graph, label, line, option group, rectangle, text box, unbound object frame).

Description

Specifies the width of a control's border.

Setting

Set this property using the property sheet or the Palette.

The BorderWidth property settings are:

Setting	Description
Hairline	(Default) The narrowest border possible on your system
1 pt–6 pt	An approximate width as indicated in points

Remarks

To use the BorderWidth property, the SpecialEffect property and the BorderStyle property must be set to Normal.

The exact border width depends on your system and printer. On some systems, the Hairline and 1 pt widths appear identical.

Access Basic

The property settings and their values are:

Setting	Value
Hairline	0
1 pt–6 pt	1–6

Design view: read/write. Other views: read-only.

See Also

BorderColor Property, BackStyle Property

BoundColumn Property

Applies To Controls (combo box, list box).

Description Binds a combo box column or a list box column to a table field specified by the ControlSource property.

Setting Enter an integer between 0 and the number set by the ColumnCount property. If you enter 0, the list index rather than a column is bound to the ControlSource property setting. The default setting is 1.

Remarks If the BoundColumn setting is 0, when you select a row in the combo box or list box while in Form view, the list index is stored in the current record. The list index of the first row is 0. If the BoundColumn setting is greater than 0, the data in the corresponding column is stored in the current record.

When a record becomes visible in Form view or in a report, Microsoft Access matches the data in the record to the bound column and then selects the appropriate row.

The leftmost visible column in a combo box contains the data that appears in the text box part of the combo box in Form view or in a report. If the LimitToList property is set to Yes, Microsoft Access searches the first column as you type text in the text box and selects the appropriate row when it finds a unique match.

Access Basic Use a numeric expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also Column Property, ColumnHeads Property, ColumnWidths Property, ControlSource Property, ListRows Property, ListWidth Property

Call Statement

Description Transfers program control to an Access Basic **Sub** procedure or a dynamic-link library (DLL) procedure.

Syntax 1 **Call** *name* [(*argumentlist*)]

Syntax 2 *name* [*argumentlist*]

Remarks The **Call** statement uses these arguments.

Argument	Description
name	Name of the procedure to call
argumentlist	Variables, arrays, or expressions to pass to the procedure

You are never required to use the **Call** reserved word when calling a procedure. If you use the **Call** reserved word to call a procedure that requires arguments, the argument list must be enclosed in parentheses. If you omit the **Call** reserved word, you also must omit the parentheses around the argument list.

You can pass arguments to a procedure by reference or by value. Values of arguments passed by reference can be altered by the procedure when the arguments are returned; Access Basic supplies the actual address of the argument. Arguments passed by value are assigned a temporary address, and the values cannot be altered. Arguments are passed by reference unless enclosed by parentheses or declared using the **ByVal** reserved word.

When passing arguments to DLL procedures, you may want to pass arguments by value. Many DLL routines don't support all Access Basic data types and **ByVal** will attempt a conversion. You can also use the **ByVal** reserved word when you declare the DLL procedure in the Declarations section.

To pass a whole array, use the array name followed by empty parentheses. However, only array elements can be passed to DLL procedures.

See Also **Declare** Statement

Example This example shows two ways to call the MessageBeep procedure in USER.EXE, a Microsoft Windows DLL.

```
Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
Sub CallDemo ()
  Call MessageBeep(0)                    ' Call Windows procedure.
  For I = 1 To 100 : Next I              ' Insert short delay between calls.
  MessageBeep 0                          ' Call again without reserved word.
End Sub
```

Cancel Property

Applies To

Control (command button) on a form.

Description

Determines whether a command button is the Cancel button on a form.

Setting

The Cancel property settings are:

Setting	Description
Yes	The command button is the Cancel button.
No	(Default) The command button isn't the Cancel button.

Remarks

Use the Cancel property to give the user the option of canceling uncommitted changes and returning the form to its previous state.

Only one command button on a form can be the Cancel button. When the Cancel property is set to Yes for one command button, it is automatically set to No for all other command buttons on the form. When the command button's Cancel property setting is Yes and the form is the active form, the user can choose the command button by clicking the button, pressing the ESC key, or pressing ENTER when the button has the focus.

Tip For a form that supports irreversible operations, such as deletions, it's a good idea to make the Cancel button the default button. To do this, set both the Cancel property and the Default property to Yes.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

CancelEvent Action

Description Cancels the event that caused Microsoft Access to run the macro containing this action. The macro is specified by an event property, such as BeforeUpdate, OnOpen, OnClose, or OnPrint.

Remarks You typically use the CancelEvent action with the BeforeUpdate property in a validation macro. When a user enters data in a field or record, Microsoft Access runs the macro before adding the data to the database. If the data fails the validation conditions in the macro, the CancelEvent action cancels updating of the database.

Often, you might use this action with the MsgBox action, which you can use to indicate that the data has failed the validation conditions and to provide helpful information about the kind of data that can be entered.

The following tables list all the event properties whose events can be canceled by the CancelEvent action and show when the specified macro runs.

Forms:

Property	Macro runs
BeforeUpdate	Before a record is saved.
OnClose	Before a form is closed.
OnDelete	Before a record is deleted.
OnInsert	Before a new record is inserted.
OnOpen	Before a form is opened.

Controls on forms:

Property	Macro runs
BeforeUpdate	Before data in the control is saved.
OnDblClick	Before the double-click action occurs.
OnExit	Before the focus leaves the control.

Reports:

Property	Macro runs
OnOpen	Before a report is opened.

Report sections:

Property	Macro runs
OnFormat	Before the page layout is set for printing.
OnPrint	Before a section is printed.

If a control's OnDblClick property setting specifies a macro containing the CancelEvent action, the action cancels the double-click event.

You can double-click any selectable control except a subform. Double-clicking produces different effects for different controls. For example, double-clicking a text box selects the word containing the insertion point. Double-clicking a control containing an OLE object automatically runs the application used to create the object so the user can edit it.

Note If a form's OnClose property specifies a macro that runs a CancelEvent action, you won't be able to close the form. You must either correct the condition that caused the CancelEvent action to run or open the macro and delete the CancelEvent action. If the form is a modal form, you won't be able to open the macro.

See Also StopMacro Action

Access Basic

Syntax

DoCmd CancelEvent

Remarks

This action takes no arguments.

The CancelEvent action has an effect only when it is run as the result of an event. This action cancels the event.

CanGrow, CanShrink Properties

Apply To Form and report sections. Tools and controls (subform/subreport, text box).

Description

- CanGrow—determines whether the size of a section or control increases so that all of its data can be printed.
- CanShrink—determines whether the size of a section or control decreases so that all of its data can be printed without leaving blank lines.

Setting The CanGrow property settings are:

Setting	Description
Yes	The section or control grows vertically so that all of its data can be printed.
No	(Default) The section or control doesn't grow. Data that doesn't fit is clipped.

The CanShrink property settings are:

Setting	Description
Yes	The section or control shrinks vertically so that its data can be printed without leaving blank lines.
No	(Default) The section or control doesn't shrink.

Remarks Use these properties to control the appearance of printed forms and reports. With both properties set to Yes, the object automatically adjusts so any amount of data can be printed. When a control grows or shrinks, the controls below it move up or down the page.

Note If you set a control's CanGrow property to Yes, Microsoft Access automatically sets the section's CanGrow property to Yes. However, if you set a control's CanShrink property to Yes, Microsoft Access does not set the section's CanShrink property to Yes. You can override this behavior by manually setting the section's CanGrow property to No.

Access Basic The settings and values for both properties are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also OnFormat Property

CanShrink Property

See CanGrow, CanShrink Properties

Caption Property

Applies To Table fields. Forms. Controls (command button, label, toggle button).

Description

- Table fields—specifies the text for labels attached to controls created by dragging a field from the field list and serves as the column heading for the field when the table is in Datasheet view.
- Forms—specifies the text that appears in the title bar in Form view.
- Buttons and labels—specifies the text that appears in the control.

Setting Enter the Caption text. Table fields, forms, and buttons can have up to 255 characters. Labels can have up to 2048 characters.

Remarks Captions are typically used to provide helpful information to the user.

If you don't set a Caption for a form, button, or label, Microsoft Access assigns the object a unique name. If you don't set a Caption for a table field, the ControlName appears in the label of a control created by dragging a field from the field list.

You can use the Caption property to assign an access key to a control. In the Caption, include an ampersand (&) immediately preceding the character you want for an access key. The character will be underlined. Press ALT plus the underlined character to move the focus to that control.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also Description Property

ChDir Statement

Description Changes the current default directory on a specified drive.

Syntax **ChDir** *path*

Remarks The argument *path* is a string expression that identifies which directory becomes the new default directory. This argument must contain fewer than 128 characters and has the following syntax:

[*drive*:] [\ ]*directory*[*directory*] . . .

The argument *drive* is an optional drive specification; the argument *directory* is a directory name. If you omit *drive*, **ChDir** changes the default directory on the current drive.

Like the operating system command **chdir**, the **ChDir** statement changes the default directory but not the default drive. For example, if the default drive is C, the following statement changes the default directory on drive D, but C remains the default drive:

```
ChDir "D:\TMP"
```

Use the **CurDir**[**$**] function to determine the current directory and the **ChDrive** statement to change the default drive.

See Also **ChDrive** Statement; **CurDir**, **CurDir$** Functions; **Dir**, **Dir$** Functions; **MkDir** Statement; **RmDir** Statement

Example This example makes the root directory the default directory on the current drive.

```
ChDir "\"
```

ChDrive Statement

Description Changes the current drive.

Syntax **ChDrive** *drive*

Remarks The argument *drive* is a string expression that specifies a new default drive. It corresponds to an existing drive and must be in the range A to *lastdrive*, where *lastdrive* is the maximum drive letter you set in your CONFIG.SYS file.

If you supply a zero-length argument (""), the current drive doesn't change. If the argument *drive* is a multiple-character string, **ChDrive** uses only the first letter.

Use the **CurDir**[**$**] function to determine the current drive and directory and the **ChDir** statement to change the default directory.

See Also **ChDir** Statement; **CurDir, CurDir$** Functions; **MkDir** Statement; **RmDir** Statement

Example This example changes the currently logged drive to the new drive indicated by the letter entered by the user.

```
NewDrive = InputBox("Enter a new drive letter.")
ChDrive NewDrive                          ' Change drive.
```

Choose Function

Description Selects and returns a value from a list of arguments.

Syntax **Choose**(*indexnum*, *varexpr* [,*varexpr*] . . .)

Remarks The **Choose** function uses the following arguments.

Argument	Description
indexnum	Numeric expression or field that results in a value of 1 to the number of *varexpr* expressions following *indexnum*.
varexpr	**Variant** expression that is a number, date, or string or a function that returns a number, date, or string. You can include a maximum of 13 expressions.

You can use **Choose** to look up one value in a list of possibilities. For example, *indexnum* might identify the number of a particular shipper. The various *varexpr* arguments would contain the names of the shippers. This might be particularly useful if *indexnum* represents the value in an option group.

Choose returns the *varexpr* determined by the value of *indexnum*. If *indexnum* is 1, **Choose** returns the first *varexpr;* if *indexnum* is 2, it returns the second one; and so on.

Choose evaluates every *varexpr,* even though it returns only one of them. For this reason, you should watch for undesirable side effects. For example, don't have a **MsgBox** function in each *varexpr* because they will all be displayed.

The **Choose** function returns a **Null** if *indexnum* is less than 1 or greater than the number of expressions in the *varexpr* list.

Following the rules for the **Fix** function, *indexnum* is rounded to the nearest whole number.

See Also **DLookup** Function, **IIf** Function, **Switch** Function

Example This example returns a string that corresponds to the name of the shipper indicated by the value of the `Ship Via` field. If `Ship Via` contains 1, **Choose** returns the first string, `"Speedy"`. If it contains 2, the function returns the second string, `"United"`, and so on.

```
= Choose([Ship Via], "Speedy", "United", "Federal")
```

In the above example, if `Ship Via` contains 0, the **Choose** function returns a **Null**. The next example shows how you can ensure that a string is returned even if the field contains 0.

```
= Choose([Ship Via] + 1, "none", "Speedy", "United", "Federal")
```

Chr, Chr$ Functions

Description Return a one-character string whose ANSI code is the argument.

Syntax **Chr**[$](*charcode*)

Remarks **Chr** returns a **Variant**; **Chr$** returns a **String**.

The argument *charcode* is an integer between 0 and 255, inclusive.

Applications for Microsoft Windows use the ANSI character set. ANSI character codes in the range 0 to 31, inclusive, are the same as the standard, nonprintable ASCII codes. For example, **Chr**(13) returns a carriage-return character, and **Chr**(10) returns a linefeed character. Together they can be used to force a new line when message strings are formatted with **MsgBox** or **InputBox**.

See Also Appendix A, "ANSI Character Set"; **Asc** Function; **Str**, **Str$** Functions

Example This example uses the **Chr** function to create a variable containing letters from A through Z. The number 64 in the argument to the **Chr** function is the ANSI code for the character preceding the letter A. Each time the line containing **Chr** is executed, another letter is added to `Alphabet`.

```
For I = 1 To 26                          ' 26 characters.
  Alphabet = Alphabet & Chr(64 + I)      ' Create a string.
Next I
```

Circle Method

Description Draws a circle, an ellipse, or an arc on a **Reports** object.

Syntax *object*.**Circle** [**Step**](*x*, *y*), *radius*[,[*color*] [,[*start*] [,[*end*] [,*aspect*]]]]

Remarks The **Circle** method uses these arguments.

Argument	Description
object	**Reports** object on which the circle is drawn.
Step	Reserved word that specifies that the center of the circle, ellipse, or arc is relative to the current coordinates given by the CurrentX and CurrentY property settings of *object*.
(x,y)	**Single** values indicating the coordinates of the center point of the circle, ellipse, or arc. The scale properties (ScaleMode, ScaleLeft, ScaleTop, ScaleHeight, and ScaleWidth) of *object* determine the units of measure used.

Argument	Description
radius	**Single** value indicating the radius of the circle, ellipse, or arc. The scale properties (ScaleMode, ScaleLeft, ScaleTop, ScaleHeight, and ScaleWidth) of *object* determine the unit of measure used.
color	**Long** value indicating the RGB color of the circle outline. If this argument is omitted, the value of the ForeColor property is used. You can use the **RGB** function or **QBColor** function to specify the color.
start, end	**Single** values. When a partial circle or ellipse is drawn, *start* and *end* specify (in radians) the beginning and end positions of the arc. The default value for *start* is 0 radians; the default for *end* is 2 * Pi radians. The range for both is –2 Pi radians to 2 Pi radians.
aspect	**Single** value indicating the aspect ratio of the circle. The default value is 1.0, which yields a perfect (nonelliptical) circle on any screen.

When drawing a partial circle or ellipse, if *start* is negative, **Circle** draws a radius to *start* and treats the angle as positive. If *end* is negative, **Circle** draws a radius to *end* and treats the angle as positive. The **Circle** method always draws in a counterclockwise (positive) direction.

You can use this method only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

To fill a circle, set the FillColor and BackStyle properties of the report. Only a closed figure can be filled. Closed figures include circles, ellipses, and pie slices (arcs with radius lines drawn at both ends.)

When drawing pie slices, if you need to draw a radius to angle 0 (to form a horizontal line segment to the right), specify a very small negative value for *start*, rather than 0.

You can omit an argument in the middle of the syntax, but you must include the argument's comma before including the next argument. If you omit a trailing argument, don't use any commas following the last argument you specify.

The width of the line used to draw the circle, ellipse, or arc depends on the DrawWidth property setting. The way the circle is drawn on the background depends on the settings of the DrawMode and DrawStyle properties.

When you apply the **Circle** method, CurrentX and CurrentY are set to the center point specified by the arguments.

See Also BackStyle Property; CurrentX, CurrentY Properties; DrawMode Property; DrawStyle Property; DrawWidth Property; FillColor Property; ForeColor Property; **QBColor** Function; **RGB** Function; ScaleHeight, ScaleWidth Properties; ScaleLeft, ScaleTop Properties; ScaleMode Property

Example The **Circle** method draws two circles on a report named `Report1`. The **Scale** method changes the scale so that the second circle is smaller and positioned in the upper-left corner.

```
Dim Rpt As Report                          ' Dimension variable.
Set Rpt = Reports!Report1                  ' Assign variable to Report.
Rpt.ScaleMode = 3                          ' Set scale to pixels.
HorCen = Rpt.ScaleWidth / 2                ' Horizontal center point.
VerCen = Rpt.ScaleHeight / 2               ' Vertical center point.
Radius = Rpt.ScaleHeight / 3               ' Circle radius.
Rpt.Circle (HorCen, VerCen), Radius        ' Draw circle.
NewHor = Rpt.ScaleWidth * 4                ' New horizontal scale.
NewVer = Rpt.ScaleHeight * 4               ' New vertical scale.
Rpt.Scale (0,0) - (NewHor,NewVer)          ' Change to new scale.
Rpt.Circle (HorCen, VerCen), Radius        ' Draw circle.
```

Clone Method

Description Creates a duplicate recordset object that refers to the same recordset.

Syntax **Set** *dupeset* = *origset*.**Clone**( )

Remarks The **Clone** method uses the following arguments.

Argument	Description
dupeset	Name of the duplicate recordset object you want to create
origset	Name of the open recordset object you want to duplicate

Use the **Clone** method to create multiple recordset objects, each of which has its own current record. Using **Clone** doesn't change the data in the recordsets or in their underlying tables.

You might use **Clone** when you want to perform an operation on a recordset that requires multiple current records. This is faster and more efficient than creating a second **Dynaset**.

The recordset you create with **Clone** initially lacks a current record. You must use one of the Move or Find methods to make a record current before you use *dupeset*.

Bookmarks are valid between recordsets created with **Clone** but not between recordsets created with the **CreateDynaset** or **CreateSnapshot** method.

See Also Bookmark Property, **Close** Method, **CreateDynaset** Method, **CreateSnapshot** Method, **OpenTable** Method

Example This example creates a **Dynaset** based on the Orders table in the NWIND.MDB database and creates `Dupeset` by cloning `Origset`. Thereafter, each **Dynaset** has its own current record that can be moved independently of the other. It uses a bookmark to make the same record current in both **Dynasets**.

```
Dim MyDB As Database, Origset As Dynaset, Dupeset As Dynaset
Dim Placeholder As String
Set MyDB = CurrentDB()
Set Origset = MyDB.CreateDynaset("Orders")    ' Create first Dynaset.
Placeholder = Origset.Bookmark                ' Save current record position.
Set Dupeset = Origset.Clone()                 ' Create duplicate Dynaset.
Dupeset.Bookmark = Placeholder                ' Go to same record.
```

Close Action

Description Closes either a specified Microsoft Access window or the active window if none is specified.

Action argument	Description
Object Type	The type of object whose window you want to close. Select Table, Query, Form, Report, Macro, or Module from this argument's drop-down list in the Action Arguments section of the Macro window. To select the active window, leave this argument blank.
Object Name	The name of the object to be closed. The drop-down list shows all objects in the database of the type selected with the Object Type argument. If you leave Object Type blank, leave this argument blank also.

Remarks The Close action works on all database objects that the user can explicitly open or close. This action has the same effect as selecting an object and then closing it using the Close command on the File menu or the Close command on the Control menu for the object.

If the object hasn't already been saved before the Close action is run, a dialog box prompts the user to save the object before the macro closes it. If you've set the Warnings On argument of the SetWarnings action to No, the dialog box doesn't appear and the object is automatically saved.

See Also Quit Action

Access Basic

Syntax

DoCmd Close [*objecttype*, *objectname*]

Argument	Description
objecttype	One of the following intrinsic constants: A_TABLE A_QUERY A_FORM A_REPORT A_MACRO A_MODULE
objectname	A string expression that is the valid name of an object of the type selected with the *objecttype* argument

Remarks

Using Close with no arguments closes the active window.

Example

This example uses the Close action to close `Order Review`.

```
DoCmd Close A_FORM, "Order Review"
```

Close Method

Description Closes a specified database, recordset, or **QueryDef** object.

Syntax *object*.**Close**

Remarks The **Close** method uses the following argument.

Argument	Description
object	Name of an open database, recordset, or **QueryDef** object

References to a closed database or its recordsets are invalid and produce an error.

Caution You should use **Close** on all open recordsets before you close a database. Any unsaved changes are lost when you close a database or recordset. Use the **Update** method to save your changes.

If the database, recordset, or **QueryDef** object named by *object* is already closed when you use **Close**, an error occurs.

If *object* refers to the current database, using the **Close** method closes the database object only. The database itself remains open. To close the database, choose Close from the File menu.

See Also **CreateDynaset** Method, **CreateQueryDef** Method, **CreateSnapshot** Method, **OpenDatabase** Function, **OpenQueryDef** Method, **OpenTable** Method, **Update** Method

Example This example opens and then closes a **QueryDef** and a table in the NWIND.MDB database.

```
Dim MyDB As Database, MyQuery As QueryDef, MyTable As Table
Set MyDB = CurrentDB()
Set MyQuery = MyDB.OpenQueryDef("Supplier List")     ' Open QueryDef.
Set MyTable = MyDB.OpenTable("Customers")            ' Open table.
...
MyQuery.Close                                        ' Close QueryDef.
MyTable.Close                                        ' Close table.
```

Close Statement

Description Concludes input/output to a file.

Syntax **Close** [[#]*filenumber*] [,[#]*filenumber*] . . .

Remarks The argument *filenumber* is the number used in the **Open** statement to open the file. You can use any numeric expression as long as it evaluates to the number of an open file.

A **Close** statement with no arguments closes all open files. However, this statement affects only files opened from within Access Basic; it doesn't affect files opened directly by Microsoft Access.

The association of a file with *filenumber* ends when a **Close** statement is executed. You then can reopen the file using the same or a different file number, or you can reuse the file number to open a different file.

Using a **Close** statement for a file that was opened for **Output** or **Append** writes the final buffer of output to the operating system buffer for that file. **Close** releases all Access Basic buffer space associated with the closed file.

The **Close** statement complements the **Open** statement.

See Also **End** Statement, **Open** Statement, **Reset** Statement, **Stop** Statement

Example This example uses **Close** to close all three files opened for **Output**.

```
For I = 1 To 3
  FNum = FreeFile                          ' Determine next file number.
  FName = "TEST" & FNum & ".DAT"
  Open FName For Output As FNum            ' Open file.
  Print #I, "This is a test."              ' Write string to file.
Next I
Close                                      ' Close all files.
```

Column Property

Applies To Tools and controls (combo box, list box) on a form.

Description Refers to a specified column of the selected item in a combo box or list box.

Setting The Column property setting is a **Variant** of **VarType** 8 (**String**).

Usage Design view: read-only. Other views: read-only.

Remarks You can use the Column property to refer to a particular column in a multiple-column combo box or list box. Use 0 to refer to the first column, 1 to refer to the second column, and so on. The following example uses the Column property to refer to the second column in a list box.

```
Debug.Print Forms![My Form]![My List Box].Column(1)
```

Note To determine how many columns a combo box or list box has, you can inspect the ColumnCount property.

You can use the Column property to assign the contents of a combo box or list box to another control, such as a text box. For example, to set the ControlSource property of a text box to the value of a column from a list box selection, you would use the following expression:

```
=Forms![My Form]![My List Box].Column(1)
```

If the user has made no selection when you refer to a column in a combo box or list box, the Column property setting will be **Null**. You can use the **IsNull** function to determine if a selection has been made:

```
If IsNull(Forms![My Form]![My List Box]) Then... ' No selection.
```

Example This example uses the Column property to print the values of a list box selection. Notice that if there is no selection, `My List Box` will be empty.

```
Sub Read_ListBox ()
  Dim NumColumns As Integer, X As Integer
  Dim F As Form
  Set F = Forms![My Form]
  If Not IsEmpty(F![My List Box]) Then              ' Any selection?
    NumColumns = F![My List Box].ColumnCount
    Debug.Print "The list box contains"; NumColumns;
    ➥ Left("columns", 6 - (NumColumns <> 1)); "."
    Debug.Print "The current selection contains:"
    For X = 1 To NumColumns
      Debug.Print F![My List Box].Column(X)         ' Print column data.
    Next X
  Else
    Debug.Print "You haven't selected an entry in the list box."
End If
End Sub
```

ColumnCount Property

Applies To Controls (combo box, graph, list box, unbound object frame [embedded]).

Description Specifies the number of columns displayed in a combo box or list box, or sent to an OLE object in a graph or object frame.

Setting Enter an integer between 1 and the maximum number of fields specified by the table or query output of the RowSource property.

Remarks A combo box or list box can have several columns. For example, if you set the ColumnCount property for a list box on an Employees form to 3, one column can list last names, another can list first names, and the third can list employee ID numbers.

If the form's RowSource property contains the name of a table or query, a combo box or list box displays the first field or query, from left to right, up to the number specified by ColumnCount. To display a different combination of fields, create either a new query or an SQL statement for the RowSource property, specifying the fields and order you want.

You can use the ColumnWidths property to set the width of the columns.

Access Basic

Use a numeric expression to set the value of this property.

Design view: read/write. Other views: read-only.

You can read a specific item in a multiple-column selection using the following syntax:

```
A = Forms!MyForm!MyList.Column(X)
```

In this example, the form name is `MyForm`, the list box name is `MyList`, and the column number is represented by `X`.

See Also

BoundColumn Property, Column Property, ColumnHeads Property, ColumnWidths Property, ControlSource Property

ColumnHeads Property

Applies To

Controls (combo box, graph, list box, unbound object frame [embedded]).

Description

Determines whether the field names in a combo box or list box are used as column headings.

Setting

The ColumnHeads property settings are:

Setting	Description
Yes	The field names are used as column headings or graph labels.
No	(Default) The field names are not used as column headings or graph labels.

Remarks

Use this property to create first-row captions for list boxes, combo boxes, and OLE objects that accept column headings. You can also use this property to create a label for each entry in a graph.

The RecordSource property specifies whether field names or the first data items are used to create column headings. For example, if a list box has three columns, the first three items of data are used as headings.

Note Headings in combo boxes appear only when the list drops down.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also

BoundColumn Property, ColumnWidths Property, ControlSource Property, ListRows Property, ListWidth Property

ColumnWidths Property

Applies To

Controls (combo box, list box).

Description

Specifies the column widths in a multiple-column combo box or list box.

Setting

Enter the column widths separated by semicolons. A width of 0 hides a column. Any or all of the ColumnWidths settings can be blank.

Remarks

If left blank, the default width is approximately 1 inch or 3 centimeters, depending on the unit of measurement.

If the columns are too wide to fully display within the combo box or list box, the rightmost columns are clipped and a horizontal scroll bar appears.

The following are sample entries for the ColumnWidths property and the resulting column widths in a system in which the default unit of measurement is inches and the list is 4 inches wide.

Setting	Widths
2;0;2	First column is 2 inches, second column is hidden, third column is 2 inches.
2;;2	First column is 2 inches, second column is 1 inch (default), third column is 2 inches. Because only half of the third column is visible, a horizontal scroll bar appears.
(Blank)	First two columns are each 1 inch (default), third column is 2 inches and uses the remaining horizontal space.

Access Basic

Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also BoundColumn Property; ColumnCount Property; ColumnHeads Property; ControlSource Property; Height, Width Properties; ListRows Property; ListWidth Property

Command, Command$ Functions

Description Return the argument portion of the command line used to launch Microsoft Access.

Syntax **Command**[$][( )]

Remarks **Command** returns a **Variant**; **Command$** returns a **String**. When **Command**[$] is used in anything except Access Basic code in a module, the empty parentheses must be included.

When Microsoft Access is launched from the command line, any portion of the command line that follows /cmd is passed to the program as the command-line argument. In the following example, `cmdlineargs` represents the argument information returned by the **Command**[$] function.

```
ACCESS /cmd cmdlineargs
```

In the Microsoft Access Module window, you can change the text returned by **Command**[$] by choosing Modify Command$ from the Run menu.

Example This example uses **Command** to display the contents of the command line.

```
If Command = "" Then                        ' If no command line.
  Msg = "There is currently no command-line string."
Else                                        ' Put command line into message.
  Msg = "The command-line string is: '" & Command & "'"
End If
MsgBox Msg                                  ' Display message.
```

CommitTrans Statement

Description Saves the current transaction and ends the transaction.

Syntax **CommitTrans**

Remarks A transaction is a series of changes you make to a recordset. You mark the beginning of a transaction with the **BeginTrans** statement. You use the **Rollback** or **CommitTrans** statement to end a transaction.

You can use **Rollback** to undo and **CommitTrans** to save changes made during the current transaction. After you make changes to your database, you can decide whether to undo the changes or commit (save) them. If you nest a transaction, you must roll back or save the current transaction before you can roll back or save one at an earlier level.

You can nest up to five levels of transactions by using multiple **BeginTrans** statements. Once you begin a transaction, use additional **BeginTrans** statements to isolate smaller groups of changes. You can then roll back or save changes to these smaller groups instead of to an entire larger transaction. If you use **BeginTrans** after five levels of transactions have already been started, an error occurs.

Once you use **CommitTrans** to save a transaction to the database, you can't undo the changes made since the most recent **BeginTrans** statement (unless the changes are part of a larger transaction that is rolled back).

If you use **CommitTrans** without a matching **BeginTrans** statement, an error occurs.

Caution If a recordset is based entirely on Microsoft Access tables, its Transactions property is **True** (–1) and you can use transactions. Tables created by other database products, however, may not support transactions. For example, you cannot use transactions in a recordset based on a Paradox table. In this case, the Transactions property is **False** (0). You should inspect the Transactions property before using the **CommitTrans** statement to make sure that the recordset supports transactions. When used with a nonsupported recordset, **CommitTrans** is ignored; no error occurs.

Note Transactions in Access Basic are independent of the transactions performed in Microsoft Access forms and reports.

See Also **BeginTrans** Statement, **Rollback** Statement, Transactions Property, **Update** Method

Example This example changes the job title of all sales representatives in the Employees table of the NWIND.MDB database. After **BeginTrans** starts a transaction that isolates all of the changes made to the Employees table, **CommitTrans** saves the changes. Notice that **Rollback** can be used to undo changes saved with the **Update** method.

```
Const MB_ICONQUESTION = 32
Const YES = 6
Const YES_NO = 4
Dim CRLF As String, EmployeeName As String
Dim Message As String, Prompt As String
Dim MyDB As Database, MyTable As Table
CRLF = Chr$(13) + Chr$(10)
Prompt = "Change title to Account Executive?"
Set MyDB = CurrentDB()                          ' Get current database.
Set MyTable = MyDB.OpenTable("Employees")       ' Open Table.
```

```
BeginTrans                                                    ' Begin transaction.
Do Until MyTable.EOF
  If MyTable!Title = "Sales Representative" Then
    EmployeeName = MyTable("Last Name") + ", " + MyTable("First Name")
    Message = "Employee: " + EmployeeName + CRLF + CRLF
    If MsgBox(Message + Prompt, MB_ICONQUESTION + YES_NO, "Change Job
    ➥ Title") = YES Then
      MyTable.Edit                                            ' Enable editing.
      MyTable!Title = "Account Executive"                     ' Change title.
      MyTable.Update                                          ' Save changes.
    End If
  End If
  MyTable.MoveNext                                            ' Move to next record.
Loop
If MsgBox("Save all changes?", MB_ICONQUESTION + YES_NO, "Save Changes")
➥ = YES Then
  CommitTrans                                                 ' Commit changes.
Else
  Rollback                                                    ' Undo changes.
End If
MyTable.Close                                                 ' Close Table.
```

Tip Using an update query to change job titles might be more efficient.

Comparison Operators

Description Used to compare two expressions.

Syntax *result* = *expr1 operator expr2*

Remarks Comparison operators, also known as relational operators, compare two expressions. The following table contains a list of the comparison operators and the conditions that determine whether *result* is **True**, **False**, or **Null**.

Operator	Meaning	True if	False if	Null if
<	Less than	*expr1* < *expr2*	*expr1* >= *expr2*	*expr1* or *expr2* = **Null**
<=	Less than or equal to	*expr1* <= *expr2*	*expr1* > *expr2*	*expr1* or *expr2* = **Null**
>	Greater than	*expr1* > *expr2*	*expr1* <= *expr2*	*expr1* or *expr2* = **Null**

Operator	Meaning	True if	False if	Null if
>=	Greater than or equal to	*expr1* >= *expr2*	*expr1* < *expr2*	*expr1* or *expr2* = **Null**
=	Equal to	*expr1* = *expr2*	*expr1* <> *expr2*	*expr1* or *expr2* = **Null**
<>	Not equal to	*expr1* <> *expr2*	*expr1* = *expr2*	*expr1* or *expr2* = **Null**

When comparing two expressions, you may not be able to easily determine whether the expressions are being compared as numbers or as strings. The following table shows how the expressions are compared or what results when either expression is not a **Variant**.

If	Then
Both expressions are numeric data types (**Integer**, **Long**, **Single**, **Double**, or **Currency**)	Perform a numeric comparison.
Both expressions are **String**	Perform a string comparison.
One expression is a numeric data type and the other is a **Variant** of **VarType** 2–7 (a numeric data type) or **VarType** 8 (**String**) that can be converted to a number	Perform a numeric comparison.
One expression is a numeric data type and the other is a **Variant** of **VarType** 8 (**String**) that can't be converted to a number	The numeric expression is less than the **String** expression.
One expression is a **String** and the other is a **Variant** of **VarType** 8 (**String**)	Perform a string comparison.
One expression is Empty and the other is a numeric data type	Perform a numeric comparison.
One expression is Empty and the other is a **String**	Perform a string comparison.
One expression is a numeric data type and the other is a **Variant** of **VarType** 8 (**String**)	A `Type mismatch` error occurs.

If *expr1* and *expr2* are both **Variant** expressions, their **VarType** determines how they are compared. The following table shows how the expressions are compared or what results from the comparison, depending on the **VarType** of the **Variant**.

If	Then
Both **Variant** expressions are of **VarType** 2–7 (numeric data types)	Perform a numeric comparison.
Both **Variant** expressions are of **VarType** 8 (**String**)	Perform a string comparison.
One **Variant** expression is of **VarType** 2–7 (a numeric data type) and the other is of **VarType** 8 (**String**)	The numeric expression is less than the **String** expression.
One **Variant** expression is Empty and the other is of **VarType** 2–7 (a numeric data type)	Perform a numeric comparison.
One **Variant** expression is Empty and the other is of **VarType** 8 (**String**)	Perform a string comparison.
Both **Variant** expressions are Empty	The expressions are equal.

Note If a **Currency** is compared with a **Single** or **Double**, the **Single** or **Double** is converted to a **Currency**. This causes any fractional part of the **Single** or **Double** value less than 0.0001 to be lost and may cause two values to compare as equal when they are not. Such a conversion can also cause an `Overflow` error if the magnitude of the **Single** or **Double** is too large.

See Also **Like** Operator, Operator Precedence, **Option Compare** Statement, **VarType** Function

Example This example shows a typical use of a comparison operator to evaluate the relationship between variables `A` and `B`. An appropriate message prints depending on whether the expression `A <= B` is **True** (–1) or **False** (0). The other comparison operators can be used in a similar way.

```
If A <= B Then
  Debug.Print "A is less than or equal to B."
Else
  Debug.Print "A is greater than B."
End If
```

Const Statement

Description Declares symbolic constants for use in place of values.

Syntax **[Global] Const** *constantname* = *expression* [,*constantname* = *expression*] . . .

Remarks The **Const** statement uses these arguments.

Argument	Description
Global	Reserved word that can precede **Const** to declare constants that can be referred to by all procedures in all modules.
constantname	Name of the constant.
expression	Expression that is assigned to the constant. It can consist of literals (such as 1.0), other constants, or any of the arithmetic or logical operators except exponentiation (^). You also can use a single string literal, such as `"Error on input"`. You cannot use string concatenation, variables, user-defined functions, or intrinsic Access Basic functions (such as **Chr**[**$**]) in expressions assigned to constants.

Tip Constants can make your programs self-documenting and easier to modify. Unlike variables, constants can't be inadvertently changed.

You can add a type-declaration character to *constantname* to indicate the data type of the constant, but this character is not part of the name. For example:

```
Const MAXDIM% = 250
...
Dim AccountNames$(MAXDIM)
```

If you don't use a type-declaration character in the name, the constant is given a data type based on the expression in the **Const** statement. Strings always yield a **String** constant, but numeric expressions are evaluated and the constant is given the simplest type that can represent the constant. Because a constant must be a single predefined type, it can't be a **Variant**. By definition, a **Variant** can appear as different data types depending on context. Constants are not affected by **Def***type* statements, such as **DefInt**.

Caution Constants must be defined before they are referred to. Make sure that **Global** constants aren't placed in multiple modules in such a way as to cause a cross-module dependency that can't be resolved. To avoid such a dependency, put your **Global** constant definitions in a single module.

Constants declared in a **Sub** or a **Function** procedure are local to that procedure. A constant declared outside a procedure (in the Declarations section of a module) is defined

throughout the module in which it is declared. Constants declared outside a procedure using the **Global** reserved word can be used by all procedures in all modules. You can use constants anywhere that you would use an expression.

Tip Use all uppercase letters for constant names to make them easy to recognize in your program listings.

See Also **Def***type* Statements

Example **Const** is used to define the symbolic constant `PI`.

```
Const PI = 3.141592654
```

ControlName Property

Applies To Controls (all).

Description Specifies the name of a control.

Setting Enter a valid name. The name can't duplicate any other ControlName on the form or report. If you don't name a control, Microsoft Access assigns it a unique name.

Remarks Microsoft Access identifies a control by its ControlName. You can use the ControlName in expressions, macros, and procedures.

If you create a control by dragging a field from the field list, the field's FieldName is copied to the ControlName.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also Caption Property

ControlSource Property

Applies To

Controls (bound object frame, check box,* combo box, list box, option button,* option group, text box, toggle button*).

* Except when the control is in an option group.

Description

Binds the control to a table field or sets an expression that creates data to display in the control.

Setting

In the property sheet, select a field from the list, type a field name, or type an equal sign (=) followed by an expression. The field name must refer to a field in the source specified by the RecordSource property. If you type an expression, the control displays the value of the expression as read-only.

Remarks

Forms and reports act as windows into your database. You must specify the location within your database of the data that you want to see in a form or report and identify the part of the data to be displayed in each control. Use the RecordSource property to specify where the data is and use the ControlSource property to specify what appears in each control.

The following table shows ControlSource settings and data interactions.

Setting	Data interaction
A field name	The control is bound to a field in a table. Data from the field displays in the control. Changes to the data inside the control change the corresponding data in the field.
An expression	The expression generates data for the control. For example, =Now( )+7 displays a date seven days from today. This data is read-only and is not saved in the database.

Caution

If a control on a form is bound to a table field, making changes to data in the control while in Form view alters the data in the table. To make the control read-only, set the Locked property to Yes.

Access Basic

Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also

RecordSource Property; RowSourceType, RowSource Properties

CopyObject Action

Description

Copies the selected database object to a different Microsoft Access database or to the same database under a new name.

Action argument	Description
Destination Database	A valid path and file name for the destination database. Leave this argument blank to select the current database.
New Name	A new name for the object. When copying to a different database, leave this argument blank to keep the same name.

Remarks

You can use this action to do the following:

- Duplicate or back up an existing object in another database.
- Quickly create a similar object with only one or two changes.

The CopyObject action is similar to selecting an object in the Database window, choosing the Copy command from the Edit menu, and then choosing Paste from the Edit menu. The Paste As dialog box appears so the user can give the object a new name. The CopyObject action performs all of these steps automatically.

You can also copy an open object in a Table window, Query window, Form window, Report window, Macro window, or Module window by choosing the Save As command from the File menu. This creates a copy of the object with a new name. If the object was saved previously, the original version still exists with the old name.

The path and file name of the destination database must exist before the macro runs the CopyObject action. If they don't exist, Microsoft Access displays an error message.

See Also

Rename Action, TransferDatabase Action

Access Basic

Syntax

DoCmd CopyObject [*destinationdatabase*] [, *newname*]

Argument	Description
destinationdatabase	A string expression that is the valid path and file name for the database you want to copy the object into. To select the current database, leave this argument blank.
newname	A string expression that is the new name for the object selected in the Database window. To use the same name if you are copying into another database, leave this argument blank.

Remarks

You must include at least one argument for the CopyObject action.

If you specify the *newname* argument and leave the *destinationdatabase* argument blank, you must include the *destinationdatabase* argument's comma. If you leave the *newname* argument blank, don't use a comma.

Example

This example uses the CopyObject action to copy the selected object (the Employees table) and give it a new name.

```
DoCmd SelectObject A_TABLE, "Employees", TRUE
DoCmd CopyObject, "Employees Copy"
```

Cos Function

Description Returns the cosine of an angle.

Syntax **Cos**(*angle*)

Remarks The argument *angle* can be any valid numeric expression measured in radians.

Cos takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

The result lies in the range –1 to 1.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi. Pi approximately equals 3.141593.

See Also **Atn** Function, **Sin** Function, **Tan** Function

Example This example uses **Cos** to calculate the cosine of an angle with a user-specified number of degrees.

```
Pi = 4 * Atn(1)                                    ' Calculate Pi.
Degrees = InputBox("Enter an angle in degrees.")   ' Specify angle.
Radians = Degrees * (Pi / 180)                     ' Convert to radians.
Cosine = Cos(Radians)                              ' Calculate cosine.
```

Count Function

Description Calculates the number of selected records in a query, form, or report.

Syntax **Count**(*expr*)

Remarks The **Count** function uses the following argument.

Argument	Description
expr	String expression identifying the field that contains the data you want to count, or an expression that performs a calculation using the data in the field. Operands in *expr* can include the name of one or more table fields, controls on a form, constants, or functions (which can be either intrinsic or user-defined but not other SQL aggregate or domain aggregate functions). You can count any kind of data, including text.

You can use **Count** to count the number of records in an underlying query. For example, you could use **Count** to count the number of orders shipped to a particular country.

Although *expr* can perform a calculation on a field, **Count** simply tallies the number of records. It doesn't matter what values are stored in the records.

The **Count** function doesn't count records that have **Null** fields unless *expr* is the wildcard character (*). If you use the wildcard character, **Count** calculates the total number of records, including those that contain **Null** fields. Do not enclose the wildcard character in quotation marks. The following example calculates the number of records in the Orders table.

```
SELECT Count(*) FROM Orders
```

If *expr* identifies multiple fields, the **Count** function counts a record if at least one of the fields is not **Null**. If all of the specified fields are **Null**, the record isn't counted. Separate the field names with an ampersand (**&**). The following example shows how you can limit the count to records in which either `Order Amount` or `Freight` is not **Null**.

```
SELECT Count("[Order Amount] & [Freight]") FROM Orders
```

You can use **Count** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **DCount** Function; **First**, **Last** Functions; **Min**, **Max** Functions; RecordCount Property; **StDev**, **StDevP** Functions; **Sum** Function; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to calculate the number of orders shipped to the United Kingdom. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT Count([Ship Country]) FROM Orders WHERE [Ship Country] = 'UK';
```

The next example calculates the number of orders for all of the underlying records in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition that limits the count to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= Count([Ship Country])
```

Count Property

Applies To Forms. Reports. **Forms** object. **Reports** object.

Description

- Forms and reports—indicates the number of controls on a form or report.
- **Forms** and **Reports** objects—indicates the number of open forms or reports.

Setting To determine the number of controls on a form or report and to assign this number to a variable, use the following syntax:

```
W = Forms![My Form].Count
X = Reports![My Report].Count
```

To determine the number of open forms or reports and to assign this number to a variable, use the following syntax:

```
Y = Forms.Count
Z = Reports.Count
```

Usage Design view: read-only. Other views: read-only.

The value of this property is an **Integer**.

Remarks If no forms or reports are open, the Count property is 0.

See Also **Forms** Object, **Reports** Object

Example This example uses the Count property to control a loop that prints information about all open forms.

```
Sub Print_Form_Controls ()
  Const MB_ICONEXCLAMATION = 48
  Dim F As Form, I As Integer, J As Integer, T As String
  Dim NumberControls As Integer, NumberForms As Integer
  T = Chr(9)                                          ' Tab character.
  NumberForms = Forms.Count                           ' Number of open forms.
  If NumberForms > 0 Then
    For I = 0 To NumberForms - 1
      Set F = Forms(I)
      Debug.Print F.FormName                          ' Print form name.
      NumberControls = F.Count
      If NumberControls > 0 Then
        For J = 0 To NumberControls - 1
          Debug.Print T; F(J).ControlName             ' Print control name.
        Next J
      Else
        Debug.Print T; "(no controls)"
      End If
    Next I
  Else
    MsgBox "There are no open forms.", MB_ICONEXCLAMATION, "Form Controls"
  End If
End Sub
```

CreateDynaset Method

Description Creates a **Dynaset** object from a specified table, **QueryDef**, or SQL string.

Syntax **Set** *dynaset* = *object*.**CreateDynaset**([*source* [, *exclusive* [, *inconsistent*]]])

Remarks The **CreateDynaset** method uses the following arguments.

Argument	Description
dynaset	Name of the **Dynaset** object you want to create.
object	Name of a database, recordset, or **QueryDef** object (except **Snapshots** created with one of the List methods).
source	String expression that is the name of an existing table or query or an SQL string. Use this argument only if *object* refers to a database object.

Argument	Description
exclusive	Boolean value that is **True** (–1) if the underlying tables are opened for exclusive (nonshared) access or **False** (0) if the tables are opened for shared access. If this argument is omitted, the tables are opened for shared access.
inconsistent	Boolean value that is **True** (–1) if the **Dynaset** will be inconsistent or **False** (0) if it will be consistent. If this argument is omitted, a consistent **Dynaset** is created.

Changes to a **Dynaset's** underlying tables are reflected in the **Dynaset** itself. Conversely, changes you make to the **Dynaset** are automatically used to update the underlying tables.

In a datasheet, Microsoft Access displays a record that has been deleted from an underlying table in the **Dynaset** as a deleted record with its values set to **Null**. A record added to an underlying table, however, doesn't appear in the **Dynaset**.

When applied to a **Dynaset** or **Snapshot**, **CreateDynaset** returns a second **Dynaset** that contains records filtered and sorted according to the **Dynaset's** or **Snapshot's** Filter and Sort properties. It doesn't affect the contents of the first **Dynaset** or **Snapshot**.

To create a **Dynaset** from a **QueryDef** that has parameters, open the **QueryDef**, set the parameters, and then use **CreateDynaset**.

Note If your program requires two identical **Dynasets**, it is more efficient to create a single **Dynaset** and then use the **Clone** method to create a second reference.

See Also **Clone** Method, **CreateSnapshot** Method, Filter Property, **OpenTable** Method, Sort Property

Example This example uses a **Dynaset** to change the title of all sales representatives in the Employees table of the NWIND.MDB database.

```
Dim MyDB As Database, MySet As Dynaset, MyTable As Table
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("SELECT * FROM Employees
➥ WHERE Title = 'Sales Representative';")
Do Until MySet.EOF
  MySet.Edit                                ' Enable editing.
  MySet!Title = "Account Executive"         ' Change title.
  MySet.Update                              ' Save changes.
  MySet.MoveNext                            ' Move to next record.
Loop                                        ' End of loop.
```

Tip Using an update query to change job titles might be more efficient.

The next example uses an SQL statement to create a **Dynaset** from the Customers table in the NWIND.MDB database. The **Dynaset** includes only customers located in the United Kingdom.

```
Dim MyDB As Database, MySet As Dynaset
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("SELECT * FROM Customers
➥  WHERE [Country] = 'UK';")
```

CreateQueryDef Method

Description Creates a new **QueryDef** (an object containing the SQL statement that describes a query) in a specified database.

Syntax **Set** *querydef* = *database*.**CreateQueryDef**(*name* [, *sqltext*])

Remarks The **CreateQueryDef** method uses the following arguments.

Argument	Description
querydef	Name of the **QueryDef** object you want to create.
database	Name of the open database object that will contain the new **QueryDef**.
name	String expression that is the name of the new **QueryDef**.
sqltext	String expression (a valid SQL statement) that defines the **QueryDef**. If you omit this argument, you can define the **QueryDef** by setting its SQL property.

The **QueryDef** is automatically saved when you create or change it, even if it wasn't defined by *sqltext*.

If you use a **QueryDef** that wasn't defined by *sqltext* or the SQL property setting, an error occurs.

To execute a **QueryDef**, use the **Execute** method. To open a **QueryDef** for changes, use the **OpenQueryDef** method.

To delete an existing **QueryDef**, use the **DeleteQueryDef** method.

See Also **Close** Method, **DeleteQueryDef** Method, **Execute** Method, **OpenQueryDef** Method

Example

Using the Customers table in the NWIND.MDB database, this example creates and defines the `All Cust` **QueryDef** in one step.

```
Dim MyDB As Database, MyQuery As QueryDef, MySet As Dynaset
Set MyDB = CurrentDB()
' Create QueryDef.
Set MyQuery = MyDB.CreateQueryDef("All Cust", "SELECT * FROM Customers;")
Set MySet = MyDB.CreateDynaset("All Cust")
...
MyQuery.Close                                          ' Close QueryDef.
MyDB.DeleteQueryDef("All Cust")                        ' Delete QueryDef.
```

The next example creates the same **QueryDef** on the same table using **CreateQueryDef** and the SQL property. The output is identical to that of the previous example.

```
Dim MyDB As Database, MyQuery As QueryDef, MySet As Dynaset
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("All Cust")          ' Create QueryDef.
MyQuery.SQL = "SELECT * FROM Customers;"               ' Set SQL property.
Set MySet = MyDB.CreateDynaset("All Cust")
...
MyQuery.Close                                          ' Close.QueryDef.
MyDB.DeleteQueryDef("All Cust")                        ' Delete QueryDef.
```

CreateSnapshot Method

Description

Creates a **Snapshot** object from a specified table, **QueryDef**, or SQL string.

Syntax

Set *snapshot* = *object*.**CreateSnapshot(** [*source*] **)**

Remarks

The **CreateSnapshot** method uses the following arguments.

Argument	Description
snapshot	Name of the **Snapshot** object you want to create.
object	Name of an open database, recordset, or **QueryDef** object (except **Snapshots** created with one of the List methods).
source	String expression that is the name of an existing table, query, or SQL string. Use this argument only if *object* is the name of a database object.

A **Snapshot** is an exact copy of the data in *source* at the point in time when you use **CreateSnapshot**.

Create a **Snapshot** to capture data at an exact moment. For example, you might want to print a year-to-date sales report containing a column of sales amounts that must match the column total. If you use a **Dynaset** (in a multiuser environment), another user could change the detail before you finish the report. In that case, the total might not agree with the detail. If you use a **Snapshot**, however, the numbers will total properly.

Unlike the data in a **Dynaset**, **Snapshot** data isn't linked to data in the underlying tables. The **Snapshot** data doesn't change as its underlying tables change. Further, data in a **Snapshot** is static (that is, it cannot be changed).

To use **CreateSnapshot** on a **QueryDef** that has parameters, first open the **QueryDef** and set the parameters.

When applied to a **Dynaset** or **Snapshot**, **CreateSnapshot** returns a **Snapshot** that contains records filtered and sorted according to the **Dynaset's** or **Snapshot's** Filter and Sort properties. The contents of the original **Dynaset** or **Snapshot** aren't affected.

When applied to a **QueryDef**, **CreateSnapshot** executes the **QueryDef** and returns a **Snapshot** that is a copy of the data defined by the **QueryDef**. If you use **CreateSnapshot** on an action **QueryDef**, an error occurs (use the **Execute** method instead).

Note You can also create a **Snapshot** using the List methods (**ListFields**, **ListIndexes**, **ListParameters**, and **ListTables**). You cannot, however, use the Find methods with a **Snapshot** created using these methods.

See Also **CreateDynaset** Method, **ListFields** Method, **ListIndexes** Method, **ListParameters** Method, **ListTables** Method

Example This example creates a **Snapshot** that contains records from the Customers table in the NWIND.MDB database. It uses the Filter property to limit the **Snapshot** to customers located in the United Kingdom. The Sort property orders the records according to the value in the Company Name field.

```
Dim MyDB As Database, MySet As Dynaset, FilteredSet As Snapshot
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Customers")          ' Create Dynaset.
MySet.Filter = "[Country] = 'UK'"                    ' Set filter.
MySet.Sort = "[Company Name]"                        ' Set sort order.
Set FilteredSet = MySet.CreateSnapshot()             ' Create Snapshot.
```

The next example (also from the Customers table of the NWIND.MDB database) shows how you can create the same **Snapshot** more efficiently using an SQL statement.

```
Set MySet = MyDB.CreateSnapshot("SELECT * FROM Customers
➥ WHERE Country = 'UK' ORDER BY [Company Name];")
```

The next examples show other ways to create a **Snapshot**, again using the NWIND.MDB database. The first uses a table name, and the second uses the name of a **QueryDef**.

```
Set MySet = MyDB.CreateSnapshot("Orders")
Set MySet = MyDB.CreateSnapshot("Large Orders")
```

CurDir, CurDir$ Functions

Description Return the current path for the specified drive.

Syntax **CurDir**[$] [(*drive*)]

Remarks **CurDir** returns a **Variant**; **CurDir$** returns a **String**. When **CurDir**[$] is used in anything except Access Basic code in a module, the empty parentheses must be included if no drive is specified.

The argument *drive* is a string expression that specifies a drive in the range A to *lastdrive*, where *lastdrive* is the maximum drive letter you set in your CONFIG.SYS file.

If no drive is specified or if *drive* is zero-length, **CurDir**[$] returns the path for the current drive. This is similar to using the **chdir** command at the system prompt without specifying a path.

An error occurs if the first character in *drive* lies outside the range A to *lastdrive*. An error also occurs if the first character in *drive* is not a letter or if it corresponds to a drive that doesn't exist.

See Also **ChDir** Statement, **ChDrive** Statement, **MkDir** Statement, **RmDir** Statement

Example This example uses **CurDir** to return the name of the current directory.

```
MsgBox "The current directory is: " & CurDir
```

Currency Data Type

Description **Currency** variables are stored as 64-bit numbers (8 bytes) in a two's complement integer format and scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range of –922,337,203,685,477.5808 to 922,337,203,685,477.5807. The type-declaration character for a **Currency** variable is @ (ANSI character 64).

The **Currency** data type is extremely useful for calculations involving money and for fixed-point calculations in which accuracy is particularly important.

See Also Access Basic Data Types

CurrentDB Function

Description Returns a database object for the current database.

Syntax **CurrentDB( )**

Remarks When you want to use Access Basic code to directly access your data, you must use a database. Usually, the current database is the one you want to use, and **CurrentDB** provides the necessary access (if you need to open a specific database, use the **OpenDatabase** function instead). The **CurrentDB** function can be used only in Access Basic code.

CurrentDB provides a way to create reusable code that doesn't depend on your knowing which database is in use at run time. You assign the value returned by **CurrentDB** to a database object that can then be used wherever a database argument is required.

CurrentDB doesn't open a database; rather, it identifies the current database that is already open. If you don't use **CurrentDB** but still want to access the current database, use **OpenDatabase**. To be successful, the current database must have been opened for nonexclusive access. If it was opened for exclusive (nonshared) access by another user, **OpenDatabase** won't work. You can still use **CurrentDB**, however.

You can create multiple database objects that each refer to the same database. In the following example, `MyDB1` and `MyDB2` both refer to the current database.

```
Dim MyDB1 As Database, MyDB2 As Database
Set MyDB1 = CurrentDB()
Set MyDB2 = CurrentDB()
```

See Also **Close** Method, **OpenDatabase** Function

Example This example uses the **CurrentDB** function to refer to the current database (in this case, NWIND.MDB) and prints an alphabetical list of company names from the Customers table. Because the name of the database doesn't appear in the code, you could use this example with other databases that have Customers tables without making any changes.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()                          ' Return database object.
Set MyTable = MyDB.OpenTable("Customers")       ' Open Table.
MyTable.Index = "Company Name"                  ' Set index.
Do Until MyTable.EOF
  Debug.Print MyTable![Company Name]            ' Print customer name.
  MyTable.MoveNext                              ' Move to next record.
Loop
MyTable.Close                                   ' Close Table.
```

CurrentX, CurrentY Properties

Apply To Forms. Reports.

Description Specify the horizontal (CurrentX) and vertical (CurrentY) coordinates for printing and drawing.

Setting Use a numeric expression to set the value of each property. When these properties are read, the value is a **Single** data type.

Usage Design view: not available. Other views: read/write.

Remarks CurrentX and CurrentY coordinates specify where the next printing or drawing method begins. The coordinates are measured from the upper-left corner of a report section. CurrentX is 0 at the section's left edge and CurrentY is 0 at its top edge. Default coordinates are expressed in twips.

To specify custom coordinates, use Access Basic to set the following related properties: ScaleHeight, ScaleWidth, ScaleLeft, ScaleTop, and ScaleMode.

When you use the following graphics methods, the CurrentX and CurrentY settings are changed as indicated.

This method	Sets CurrentX, CurrentY to
Circle	The center of the object.
Line	The end point of the line.
Print	The next print position.

See Also DrawMode Property; DrawStyle Property; Left, Top Properties

CVDate Function

Description Converts an expression to a **Variant** of **VarType** 7.

Syntax **CVDate**(*expression*)

Remarks The argument *expression* must be a string expression or a numeric expression that can be interpreted as a date. The acceptable range for date information is January 1, 100 A.D. (–657434) through December 31, 9999 A.D. (2958465).

The following table identifies the action that occurs as a result of evaluating *expression*.

If *expression* is this type	CVDate takes this action
A numeric expression between –657434 and 2958465, inclusive	Considers *expression* to already be a date serial and makes no change
A numeric expression less than –657434 or greater than 2958465	Generates a run-time error
A numeric expression that looks like a date, for example, "1 March 1992 12:00 PM"	Converts *expression* to a date serial, for example, 33664.5
A numeric expression that looks like a number, for example, "33664"	Converts *expression* to a date serial, for example, 33664

For string expressions, **CVDate** recognizes all the date formats that can be set in the International section of the WIN.INI file using the Microsoft Windows Control Panel.

Notes

- You can use the **IsDate** function to determine whether a string expression can be converted to a date. You should explicitly test numeric expressions to ensure that they fall within the range of acceptable dates.
- When converting from a date serial to date format, Access Basic converts any fractional part of the numeric expression to a time of day, starting at midnight.

See Also Data Type Conversion Functions, **IsDate** Function

Example This example returns a **Variant** that contains a Date data type equal to 33,770.

```
= CVDate("15 June 1992")
```

Data Type Conversion Functions

Description Explicitly converts expressions from one data type to another.

Syntax

CCur(*expression*)

CDbl(*expression*)

CInt(*expression*)

CLng(*expression*)

CSng(*expression*)

CStr(*expression*)

CVar(*expression*)

Remarks

The argument *expression* can be any valid string expression or numeric expression.

The following table shows which data type is returned by each data type conversion function.

Function	From	To
CCur	Any valid expression	**Currency**
CDbl	Any valid expression	**Double**
CInt	Any valid expression	**Integer**
CLng	Any valid expression	**Long**
CSng	Any valid expression	**Single**
CStr	Any valid expression	**String**
CVar	Any valid expression	**Variant**

The numeric conversion functions **CCur**, **CDbl**, **CInt**, **CLng**, and **CSng** explicitly control the data type of a numeric expression. For example, you can use **CCur** to force currency arithmetic (which has greater precision but less range) in cases in which integer, double-precision, or single-precision arithmetic normally would occur. **CInt** and **CLng** force integer arithmetic in cases in which currency, single-precision, or double-precision arithmetic normally would occur. **CDbl** and **CSng** force double- or single-precision arithmetic in cases in which currency or integer arithmetic normally would occur.

Tip All data type conversion functions can provide self-documenting code that indicates that the result of a calculation should be expressed as a particular data type rather than the normal data type of the result.

For the **CCur**, **CInt**, **CLng**, and **CSng** functions, if *expression* lies outside the acceptable range, a run-time error occurs and a message is displayed unless it is trapped in error-handling code.

Note **CInt** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. For a demonstration of the differences between **Int** and **Fix**, see the example for **Int**. When the fractional part is exactly 0.5, the **CInt** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

See Also

Access Basic Data Types; **Int** Function, **Fix** Function

Example

CCur converts the calculated **Double** to a **Currency** with four decimal digits of precision to the right of the decimal place, displaying the result in conventional currency value format with two places to the right of the decimal.

```
= CCur(Amount) * CCur(Percent * .01)
```

CDbl converts a **Currency** value to a **Double**.

```
= CDbl(CurrVal)
```

CInt converts a calculated **Double** to an **Integer**.

```
= CInt(DblVal * .33)
```

CLng converts each **Single** value to a **Long**. Note that 25427.45 is rounded down to 25427, whereas 25427.55 is rounded up to 25428.

```
= CLng(25427.45)
= CLng(25427.55)
```

CSng converts each **Double** value to a **Single**. Note that 75.3421115 is rounded down to 75.34211, whereas 75.3421155 is rounded up to 75.34212.

```
= CSng(75.3421115)
= CSng(75.3421155)
```

CStr converts a **Double** to a **String**.

```
= CStr(Pi#)
```

CVar takes a **String** value expressed in thousands for sales from a quarterly report, concatenates three zeros, and converts the entire expression to a **Variant**.

```
= CVar(Sales$ & "000")
```

DataType Property

Applies To Table fields.

Description Specifies the type of data stored in a table field.

Setting The DataType property settings are:

Setting	Description
Text	(Default) Text up to 255 characters or the length set by the FieldSize property, whichever is less.
Memo	Text with a maximum length of 32,000 characters. Memo fields can't be indexed.
Number	Any numerical type. See FieldSize for more information.
Date/Time	Date and time values for years from 100 to 9999.
Currency	Accurate to 15 digits on the left side of the decimal point and four digits on the right.
Counter	A number automatically incremented by Microsoft Access whenever a new record is added to a table. Counter fields can't be updated.
Yes/No	Yes and No values and fields that contain only one of two values. Yes/No fields can't be indexed.
OLE Object	An object such as a Microsoft Excel spreadsheet or a Microsoft Draw graphic created by one of the OLE servers on your system. The maximum size is about 128 megabytes. OLE Object fields can't be indexed.

Remarks You must specify the type of data stored in each field in a table. Each field stores data of only one type.

Note Use the Currency data type for a field requiring many calculations involving data with one to four decimal places. Single and Double fields require floating-point calculation. The Currency data type uses a faster fixed-point calculation that avoids rounding errors.

Caution Changing a field data type after you enter data in a table causes a potentially lengthy process of data conversion when you save the table. If data types conflict, you may lose some data.

Access Basic This property is not available from Access Basic. To get information about the data type of a table field, use the **ListFields** method.

See Also Access Basic Data Types, FieldSize Property, Format Property, **ListFields** Method

Date, Date$ Functions

Description Return the current system date.

Syntax **Date**[$][( )]

Remarks When **Date**[$] is used anywhere except in Access Basic code in a Microsoft Access module, the empty parentheses must be included to avoid confusion with a field named Date.

Date returns a **Variant** (**VarType** 7) containing a date stored internally as a **Double**.

Date$ returns a 10-character string of the form *mm-dd-yyyy*, where *mm* is the month (01–12), *dd* is the day (01–31), and *yyyy* is the year (1980–2099).

The output of the **Date$** function is equivalent to:

```
Format$(Now,"mm-dd-yyyy")
```

To set the system date, use the **Date**[$] statement.

See Also **CVDate** Function; **Date**, **Date$** Statements; **Format**, **Format$** Functions; **Now** Function

Example This example uses **Date**[$] to return the current value of the computer system date.

```
= Date$
```

Date, Date$ Statements

Description Set the current system date.

Syntax **Date**[$] = *expression*

Remarks For the **Date** statement, *expression* must be a **String** or string variable formatted as a date or a **Variant** of **VarType** 7 (Date) or **VarType** 8 (**String**). If *expression* is a **String** (or a **Variant** of **VarType** 8) that contains only numbers delimited by valid date separators, **Date** recognizes the order for month, day, and year according to the Short Date format in the International section of your WIN.INI file. **Date** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **Date** recognizes December 30, 1991; Dec 30, 1991; 30-Dec-1991; and 30 December 91. If *expression* is not already a **Variant** of **VarType** 7, **Date** attempts to convert it. If it can't be converted to a date from January 1, 1980 through December 31, 2099, an error occurs.

For **Date$**, *expression* must have one of the following forms, where *mm* (01–12) and *dd* (01–31) are the month and day and *yy* or *yyyy* (1980–2099) is the year:

mm-dd-yy

mm-dd-yyyy

mm/dd/yy

mm/dd/yyyy

If you use **Date**, *expression* must be enclosed by number signs: `#5/30/94#`. If you use **Date$**, *expression* must be enclosed by either number signs or quotation marks: `#5/30/94#` or `"5/30/94"`.

If you use the **Date**[$] statement to set the date with versions of MS-DOS® earlier than 3.3, the change remains in effect only until you change it again or turn off your computer. Many computers have a battery-powered CMOS RAM that retains date and time information when the computer is turned off. However, to permanently change the date on computers running earlier versions of MS-DOS, you may have to use your Setup disk or perform some equivalent action. Refer to the documentation for your particular system.

See Also **Date, Date$** Functions; **Time, Time$** Functions; **Time, Time$** Statements

Example In this example, the **Date** statement sets the system date to a new date entered by the user. The **Date** function provides the default date in the input box.

```
Date = InputBox("Enter new date: ", "", Date)
```

DateAdd Function

Description Returns a **Variant** of **VarType** 7 (Date) to which a specified time interval has been added.

Syntax **DateAdd**(*interval*, *number*, *date*)

Remarks The **DateAdd** function uses the following arguments.

Argument	Description
interval	String expression that is the interval of time you want to add to *date*.
number	Numeric expression that is the number of intervals you want to add. It may be positive (to get dates in the future) or negative (to get dates in the past).
date	Date being added to, or the name of a **Variant** of **VarType** 7 (Date). It can be either short form (`1/1/95`) or long form (`1-January-1995`).

You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you might use **DateAdd** to calculate a date 30 days from today or a time that is 60 minutes from now.

The following table lists the valid time periods and their *interval* values. These intervals can also be used by the **Format** function.

Time period	*interval*
Year	`yyyy`
Quarter	`q`
Month	`m`
Day of Year	`y`
Day	`d`
Weekday	`w`
Week	`ww`
Hour	`h`
Minute	`n`
Second	`s`

If you want to add days to *date*, you can use Day of Year, Day, or Weekday (`"y"`, `"d"`, or `"w"`).

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31.

```
= DateAdd("m", 1, #31-Jan-95#)
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If *date* was 31-Jan-96, it would return 29-Feb-96 because 1996 is a leap year.

If you omit the year from *date*, **DateAdd** uses the current year. If *date* is enclosed by number signs (#), the current year becomes a permanent part of *date*. If you enclose *date* in double quotation marks, however, the current year is inserted in your code each time it is run.

If the calculated date would precede the year 0 (that is, you subtract more years than are in *date*), an error occurs.

If *number* isn't a **Long** value, **DateAdd** follows the rules for the **Fix** function and rounds *number* to the nearest whole number.

See Also **DateDiff** Function, **DatePart** Function, **Now** Function

Example The following example calculates a ship date that is 10 days before the promised arrival date.

```
ShipDate = DateAdd("y", -10, PromisedDate)
```

The next example prints a list of employees' anniversary dates. When you call this routine, you provide an argument (`Years`) that is the number of years you want to add to the employees' hire dates.

```
Sub Anniversary_List (Years As Long)
  Dim Anniversary As String, Employee As String
  Dim MyDB As Database, MyTable As Table
  Set MyDB = CurrentDB()
  Set MyTable = MyDB.OpenTable("Employees")        ' Open Employees table.
  MyTable.Index = "Last Name"                      ' Set table index.
  Do Until MyTable.EOF                             ' For every record.
    Anniversary = DateAdd("yyyy", Years, MyTable![Hire Date])
    Employee = MyTable![Last Name] & ", " & MyTable![First Name]
    Debug.Print Employee ; " "; Anniversary        ' Print name, date.
    MyTable.MoveNext                               ' Move to next record.
  Loop
  MyTable.Close                                    ' Close table.
End Sub
```

DateCreated Property

Applies To **Tables**.

Description Contains the date and time when a **Table** was created. This property can be used only in Access Basic code.

Setting The DateCreated property setting is a **Variant** of **VarType** 7 (Date).

Usage Read-only.

Remarks In a multiuser environment, the date and time are taken from the computer on which the **Table** was created. If the various users maintain different dates and times on their workstations, the DateCreated property setting could be incorrect. When possible, each user should get the date and time directly from the file server.

See Also LastUpdated Property

DateDiff Function

Description Returns a **Variant** that contains the number of time intervals between two specified dates.

Syntax **DateDiff**(*interval*, *date1*, *date2*)

Remarks The **DateDiff** function uses the following arguments.

Argument	Description
interval	String expression that is the interval of time you use to calculate the difference between *date1* and *date2*
date1, date2	Two dates you want to use in the calculation, or the names of two Date/Time fields (or a combination)

You can use the **DateDiff** function to determine how many time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between an order date and its ship date or the number of weeks between today and the end of the year.

The following table lists the valid time periods and their *interval* values. These intervals can also be used by the **Format** function.

Time period	*interval*
Year	`yyyy`
Quarter	`q`
Month	`m`
Day of Year	`y`
Day	`d`
Weekday	`w`
Week	`ww`
Hour	`h`
Minute	`n`
Second	`s`

If you want to know the number of days between *date1* and *date2*, you can use either Day of Year or Day (`"y"` or `"d"`).

The following example calculates the number of days between today and Christmas Day.

```
= DateDiff("y", Now(), "25-Dec")
```

In this case, the year is omitted from *date2*. When you omit the year, **DateDiff** uses the current year. If a date is enclosed by number signs (#), the current year becomes a permanent part of that date. If you enclose the date in double quotation marks, however, the current year is inserted in your code each time it is run. This makes it possible to write code that can be used over multiple years.

When *interval* is Weekday (`"w"`), **DateDiff** returns the number of weeks between the two dates. If *date1* falls on a Monday, **DateDiff** counts the number of Mondays until *date2*. It counts *date1* but not *date2*. If *interval* is Week (`"ww"`), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between *date1* and *date2*. **DateDiff** counts *date1* if it falls on a Sunday; but it doesn't count *date2*, even if it does fall on a Sunday.

If *date1* refers to a later point in time than *date2*, the **DateDiff** function returns a negative number.

See Also **DateAdd** Function, **DatePart** Function, **Now** Function

Example The following example calculates the number of days between an order date and a ship date.

```
LeadTime = DateDiff("y", OrderDate, ShippedDate)
```

The next example calculates the number of calendar weeks between the first of the year and today, and the number of days since July 4, 1776.

```
Elapsed = DateDiff("ww", "1-1", Now())
CountryAge = DateDiff("y", #4-Jul-1776#, Now())
```

DatePart Function

Description Returns a specified part of a given date.

Syntax **DatePart**(*interval*, *date*)

Remarks The **DatePart** function uses the following arguments.

Argument	Description
interval	String expression that is the interval of time you want to return
date	Date that you want to inspect, or the name of a Date/Time field

You can use the **DatePart** function to inspect a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week for an order's ship date or the current hour.

The following table lists the valid time periods and their *interval* values. These intervals can also be used by the **Format** function.

Time period	*interval*
Year	`yyyy`
Quarter	`q`
Month	`m`
Day of Year	`y`
Day	`d`
Weekday	`w`
Week	`ww`

Time period	*interval*
Hour	h
Minute	n
Second	s

Note For Weekday, Sunday = 1, Monday = 2, and so on.

The following example calculates which day of the week Christmas falls on.

```
= DatePart("w", "25-Dec")
```

In this case, the year is omitted from *date*. When you omit the year, **DatePart** uses the current year. If a date is enclosed by number signs (#), the current year becomes a permanent part of that date. If you enclose the date in double quotation marks, however, the current year is inserted in your code each time it is run.

The next example determines the calendar quarter in which an order was placed.

```
= DatePart("q", Forms![Orders]![Order Date])
```

See Also **DateAdd** Function, **DateDiff** Function, **Day** Function, **Hour** Function, **Minute** Function, **Month** Function, **Now** Function, **Second** Function, **Weekday** Function, **Year** Function

Example The following example determines the day of the week on which an order was placed.

```
DayOfWeek = DatePart("w", OrderDate)
```

The next example determines a birth year.

```
BirthYear = DatePart("yyyy", MyBirthDate)
```

DateSerial Function

Description Returns the date serial for a specific year, month, and day.

Syntax **DateSerial**(*year*, *month*, *day*)

Remarks The **DateSerial** function uses these arguments.

Argument	Description
year	A number between 0 and 9999, inclusive, or a numeric expression
month	A number between 1 and 12, inclusive, or a numeric expression
day	A number between 1 and 31, inclusive, or a numeric expression

To express a specific date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should conform to the accepted range of values for the unit. These values are 1 through 31 for days and 1 through 12 for months. You also can specify relative dates for each argument by using a numeric expression representing the number of days, months, or years before or after a certain date. The following example uses expressions instead of absolute date numbers. The **DateSerial** function returns a date that is the day before the first day (`1 - 1`) of two months before August (`8 - 2`) of 10 years before 1990 (`1990 - 10`)—in other words, May, 31, 1980.

```
DateSerial(1990 - 10, 8 - 2, 1 - 1)
```

For the argument *year*, values between 0 and 99, inclusive, are interpreted as the years 1900–1999. For all other *year* arguments, use the complete four-digit year (for example, 1800).

The **DateSerial** function returns a **Variant** of **VarType** 7 (Date) containing a date that is stored internally as a double-precision number. This number represents a date from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Negative numbers represent dates prior to December 30, 1899.

If the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates, an error occurs.

See Also **DateValue** Function, **Day** Function, **Month** Function, **Now** Function, **Weekday** Function, **Year** Function

Example In this example, the **DateSerial** function returns a **Variant** (**VarType** 7) containing a date created from the integer arguments.

```
= DateSerial(Y, M, D)
```

DateValue Function

Description Returns the date represented by the date of a **String** argument.

Syntax **DateValue**(*stringexpression*)

Remarks The argument *stringexpression* is a **String** representing a date from January 1, 100 through December 31, 9999.

If *stringexpression* includes only numbers, **DateValue** recognizes the order for month, day, and year according to the date setting in the International section of your WIN.INI file. **DateValue** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** recognizes December 30, 1991; Dec 30, 1991; 30-Dec-1991; and 30 December 91.

If the year part of *stringexpression* is omitted, **DateValue** uses the current year from your computer's system date.

If *stringexpression* includes time information, **DateValue** doesn't return it. However, if *stringexpression* includes invalid time information (such as `"89:98"`), an error occurs.

The **DateValue** function returns a **Variant** of **VarType** 7 (Date) containing a date that is stored internally as a double-precision number. This number represents a date from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Negative numbers represent dates prior to December 30, 1899.

See Also **DateSerial** Function, **Day** Function, **Month** Function, **Now** Function, **Weekday** Function, **Year** Function

Example In this example, the **DateValue** function converts the **String** provided by the user to a **Variant** of **VarType** 7 (Date). The **Variant** is then used to determine the day of the week for the user-provided date.

```
Msg = "Enter a date in the form mm/dd/yyyy."
UserDate = InputBox(Msg)                          ' Get user input.
If Len(UserDate) <> 10 Then                       ' Use today's date
  UserDate = Format(Now, "mm/dd/yyyy")            '  if user didn't
End If                                            '  use right format.
RelVal = DateValue(UserDate)                      ' Get date serial.
Select Case RelVal                                ' Use correct verb.
  Case Is < Int(Now): Verb = " was a "
  Case Is > Int(Now): Verb = " will be a "
  Case Else: Verb = " is a "
End Select
WhatDay = Format(RelVal, "dddd")                  ' Determine day.
MsgBox UserDate & Verb & WhatDay & "."            ' Display message.
```

DAvg Function

Description Returns the arithmetic mean of a set of values in a specified set of records (domain).

Syntax **DAvg**(*expr*, *domain* [, *criteria*])

Remarks The **DAvg** function uses the following arguments.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to average, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DAvg** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DAvg** evaluates *expr* against the entire domain.

The average calculated by **DAvg** is the arithmetic mean (the sum of the values divided by the number of values). You might use **DAvg** to calculate average freight cost.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following examples from the Orders table of the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DAvg("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DAvg("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use **DAvg** in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DAvg("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If any records in *domain* are being edited when you use **DAvg**, the function uses the most recently saved values.

Records that contain **Null** fields aren't included in the calculation.

See Also Avg Function

Example This example returns the average freight cost for large orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK` and `Order Amount` is greater than `1000`.

```
X = DAvg("[Freight]", "Orders", "[Ship Country] = 'UK' And
➥ [Order Amount] > 1000")
```

The next example calculates the same average using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DAvg("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry
➥ & "'And [Order Amount] > 1000")
```

Day Function

Description Returns an integer between 1 and 31, inclusive, that represents the day of the month for a date argument.

Syntax **Day**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Hour** Function, **Minute** Function, **Month** Function, **Now** Function, **Second** Function, **Weekday** Function, **Year** Function

Example This example uses the **Day** function to determine the day of the month.

```
DayNumber = Day(Now)                    ' Get the current day number.
Select Case DayNumber                   ' Get the proper suffix
  Case 1, 21, 31                        '  for each number.
    DaySuffix = "st"
  Case 2, 22
    DaySuffix = "nd"
```

```
    Case 3, 23
      DaySuffix = "rd"
    Case Else
      DaySuffix = "th"
End Select
MsgBox "Today is the " & DayNumber & DaySuffix & " day of the month."
```

DCount Function

Description Returns the number of selected records in a specified set of records (domain).

Syntax **DCount**(*expr*, *domain* [, *criteria*])

Remarks The **DCount** function uses the following arguments.

Argument	Description
expr	String expression identifying the field that contains the data you want to count, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of one or more table fields, controls on a form, constants, or functions (which can be either intrinsic or user-defined but not other domain aggregate or SQL aggregate functions). You can count any kind of data, including text.
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DCount** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DCount** evaluates *expr* against the entire domain.

If you don't need to know particular values, you can use **DCount** to count the number of records in a domain. For example, you could use **DCount** to count the number of orders shipped to a particular region. Although *expr* can perform a calculation on a field, **DCount** simply tallies the number of records. It doesn't matter what values are stored in the records.

Unless *expr* is the asterisk (*) wildcard character, the **DCount** function doesn't count records that contain **Null** fields. If you use an asterisk, **DCount** calculates the total number

of records, including those that contain **Null** fields. The following example calculates the number of records in the Orders table in the NWIND.MDB database.

```
X = DCount("*", "Orders")
```

If *domain* is an indexed table, you can also count the total number of records by setting *expr* to the primary key field. The primary key field in an indexed table will not be **Null**.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table of the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DCount("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DCount("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use **DCount** in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DCount("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If *expr* identifies multiple fields, separate the field names with either a plus sign (+) or an ampersand (&).

If any records in *domain* are being edited when you use **DCount**, the function uses the most recently saved values.

See Also **Count** Function, **DAvg** Function, **DSum** Function

Example This example returns the number of large orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK` and `Order Amount` is greater than `1000`.

```
X = DCount("[Order Amount]", "Orders", "[Ship Country] = 'UK' And
➥ [Order Amount] > 1000")
```

The next example returns the same number using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DCount("[Order Amount]", "Orders", "[Ship Country] = '" &
➥ SearchCountry & "'And [Order Amount] > 1000")
```

DDB Function

Description Returns the depreciation of an asset for a specific period using the double-declining balance method.

Syntax **DDB**(*cost*, *salvage*, *life*, *period*)

Remarks The double-declining balance method computes depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods.

The **DDB** function uses the following numeric arguments.

Argument	Description
cost	Initial cost of the asset
salvage	Value of the asset at the end of its useful life
life	Length of the useful life of the asset
period	Period for which asset depreciation is calculated

The arguments *life* and *period* must be expressed in the same units. For example, if *life* is given in months, *period* must also be given in months. All arguments must be positive numbers.

The **DDB** function uses the following formula:

Depreciation over *period* = ((*cost* – total depreciation from prior periods) * 2) / *life*

See Also **SLN** Function, **SYD** Function

Example Using the double-declining balance method, this example returns the depreciation of an asset for a specified period given the initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), the total life of the asset in years (`LifeTime`), and the period for which the depreciation is calculated (`Depr`), also in years.

```
Const YRMOS = 12                              ' Number of months in a year.
Fmt = "###,##0.00"
InitCost = InputBox("What's the initial cost of the asset?")
SalvageVal = InputBox("What's the asset's value at the end of its life?")
MonthLife = InputBox("What's the asset's useful life in months?")
Do While MonthLife < YRMOS                    ' Ensure period is >= 1 year.
  MsgBox "Asset life must be a year or more."
  MonthLife = InputBox("What's the asset's useful life in months?")
Loop
LifeTime = MonthLife / YRMOS                  ' Convert months to years.
If LifeTime <> Int(MonthLife / YRMOS) Then
  LifeTime = Int(LifeTime + 1)                ' Round up to nearest year.
End If
```

```
DepYear = CInt(InputBox("For what year do you want to calculate the
➥ depreciation?"))
Do While DepYear < 1 Or DepYear > LifeTime
  MsgBox "You must enter at least 1 but not more than " & LifeTime
  DepYear = InputBox("For what year do you want to calculate the
  ➥ depreciation?")
Loop
Depr = DDB(InitCost, SalvageVal, LifeTime, DepYear)
MsgBox "The depreciation for year " & DepYear & " is " & Format(Depr,
➥ Fmt) & "."
```

DDE Function

Description Initiates a dynamic data exchange (DDE) conversation with another application and requests an item of information from that application.

Syntax **DDE** (*application*, *topic*, *item*)

Remarks You can use the **DDE** function only on forms in the ControlSource property of text boxes, option groups, check boxes, and combo boxes.

The **DDE** function uses the following arguments.

Argument	Description
application	String expression that is the name of an application that can participate in a DDE conversation. Usually, this is the name of an .EXE file (without the .EXE extension) for a Microsoft Windows–based application, such as Microsoft Excel.
topic	String expression that is the name of a topic recognized by *application*.
item	String expression that is the name of a data item recognized by *application*.

If the **DDE** function successfully initiates a DDE conversation, it returns a **Variant** of **VarType** 8 (**String**) that contains the requested information. The maximum number of DDE conversations that can be open simultaneously is determined by Windows and your computer's memory and resources. If the conversation can't be initiated because *application* isn't running or doesn't recognize *topic*, or if the maximum number of conversations has already been reached, **DDE** returns a **Null**.

Note The other application may be configured to ignore your DDE conversation. If so, **DDE** returns a **Null**. Similarly, you can set Microsoft Access to ignore requests from other applications by choosing the Options command from the View menu. Then select the Ignore DDE Requests item and change its setting to Yes.

The value of *topic* depends on *application*. For applications that use documents or data files, *topic* often includes the names of those files. The value of *item* also depends on *application*. You define the topic name in the other application, not in Microsoft Access.

For Microsoft Excel, *item* might be a row and column identifier, such as "R1C1", or the name of a range of cells. In the following example, the **DDE** function requests information from the cell at row 1, column 1 in a Microsoft Excel worksheet. You can enter this expression for a text box control in the ControlSource property in the control's property sheet.

```
=DDE("Excel", "Sheet1", "R1C1")
```

When you use the **DDE** function, the control becomes read-only in Form view and Print Preview. For example, if you use the **DDE** function in a text box, the text in the text box cannot be edited. You must edit the text in the other application. Because the ControlSource property is also read-only in Form view and Print Preview, changes to the control must be made in Design view.

Tip You can also use the Locked property to make a control read-only.

The following table illustrates how the **DDE** function behaves when you use it with each of the controls.

Control	Remarks
Text box	The *item* argument can refer to text or numbers. If *item* refers to more than one piece of information, such as a named range in a Microsoft Excel worksheet that contains multiple cells, **DDE** returns the first entry.
Option group	The OptionValue property of each option button in an option group is set to a number. Usually, the first button value is 1, the second is 2, and so on. The number returned by the **DDE** function determines which option button will be selected. For example, if **DDE** returns 2, the second button will be selected. If **DDE** returns a value that doesn't match any of the Option Value settings, none of the buttons will be selected. If *item* refers to more than one piece of information, such as a named range in a Microsoft Excel worksheet that contains multiple cells, **DDE** returns the first entry.

Control	Remarks
Check box	If the **DDE** function returns 0, the check box will be cleared. If **DDE** returns a nonzero number, such as 1 or –1, the box will be selected. If *item* refers to text or to more than one piece of information, such as a named range in a Microsoft Excel worksheet that contains multiple cells, the check box will be unavailable.
Combo box	The **DDE** function fills in the combo box with the information referred to by *item*. You can't enter new data in the text portion of the box. You might use the **DDE** function with a combo box to display a list of states or countries that you maintain in a Microsoft Excel worksheet.

See Also **DDEInitiate** Function; **DDERequest** Function; **DDESend** Function; Enabled, Locked Properties; OptionValue Property

DDEExecute Statement

Description Uses an open dynamic data exchange (DDE) channel to send a command to another application.

Syntax **DDEExecute** *channum, command*

Remarks The **DDEExecute** statement uses the following arguments.

Argument	Description
channum	Channel number returned by the **DDEInitiate** function
command	String expression containing a command recognized by the other application

The actual value of *command* depends on the application and topic specified when the channel *channum* is opened. An error occurs if *channum* isn't an **Integer** corresponding to an open channel or if the other application can't perform the specified command.

See Also **DDEInitiate** Function

Example

This example establishes a DDE link with Microsoft Excel, places some values into cells in the first row of a new worksheet, and charts the values. **DDEExecute** sends Microsoft Excel the command to begin a new worksheet.

```
On Error Resume Next                              ' Set up error handler.
Dim Chan, SheetName, I, TopicList                 ' Declare variables.
Chan = DDEInitiate("Excel", "System")             ' Establish spreadsheet link.
If Err Then                                       ' If error occurs, spreadsheet
  Err = 0                                         '  isn't running. Reset error
  I = Shell("Excel", 1)                           '   and start spreadsheet.
  If Err Then Exit Sub                            ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")           ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                       ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")         ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                                 ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)            ' Establish link with new sheet.
For I = 1 To 10                                   ' Put some values into
  DDEPoke Chan, "R1C" & I, I                      '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]"       ' Make chart.
DDETerminateAll                                   ' Terminate all links.
```

DDEInitiate Function

Description

Begins a dynamic data exchange (DDE) conversation with another application.

Syntax

DDEInitiate(*application*, *topic*)

Remarks

The **DDEInitiate** function uses the following arguments.

Argument	Description
application	String expression containing the name of an application that can respond to DDE. Usually, this is the name of an .EXE file (without the .EXE extension) of an application for Microsoft Windows.
topic	String expression containing the name of a topic recognized by the application named by *application*.

If **DDEInitiate** successfully initiates a DDE conversation with the specified application and topic, it returns a **Variant** containing a channel number that can be used with other DDE functions and statements. An error occurs if the application is not already running, or if it is running but doesn't recognize *topic* or doesn't support DDE.

The actual value of *topic* depends on the application. For applications that use documents or data files, valid topic names often include the names of those files.

The maximum number of channels that can be open simultaneously is determined by Windows and your system's memory and resources. If you aren't using a channel, you should conserve resources by terminating it using a **DDETerminate** or **DDETerminateAll** statement.

See Also **DDETerminate** Statement, **DDETerminateAll** Statement

Example This example establishes a DDE link with Microsoft Excel, places some values into cells in the first row of a new worksheet, and charts the values. **DDEInitiate** begins the DDE conversation with Microsoft Excel.

```
On Error Resume Next                              ' Set up error handler.
Dim Chan, SheetName, I, TopicList                 ' Declare variables.
Chan = DDEInitiate("Excel", "System")             ' Establish spreadsheet link.
If Err Then                                       ' If error occurs, spreadsheet
  Err = 0                                         '  isn't running. Reset error
  I = Shell("Excel", 1)                           '  and start spreadsheet.
  If Err Then Exit Sub                            ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")           ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                       ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")         ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                                 ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)            ' Establish link with new sheet.
For I = 1 To 10                                   ' Put some values into
  DDEPoke Chan, "R1C" & I, I                      '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]"       ' Make chart.
DDETerminateAll                                   ' Terminate all links.
```

DDEPoke Statement

Description Uses an open dynamic data exchange (DDE) channel to supply data to another application.

Syntax **DDEPoke** *channum, item, data*

Remarks The **DDEPoke** statement uses the following arguments.

Argument	Description
channum	Channel number returned by the **DDEInitiate** function
item	String expression containing the name of a data item recognized by the other application
data	String containing the data supplied to the other application

The actual value of *item* depends on the application and topic specified when the channel *channum* is opened. The value of *data* depends on the item specified. An error occurs if *channum* isn't an **Integer** corresponding to an open channel, or if the other application doesn't recognize or accept the specified data.

The data is supplied in plain text format. No other formats are supported.

See Also **DDEInitiate** Function, **DDERequest** Function, **DDESend** Function

Example This example establishes a DDE link with Microsoft Excel, places some values into cells in the first row of a new worksheet, and charts the values. **DDEPoke** sends the data to be charted to Microsoft Excel.

```
On Error Resume Next                              ' Set up error handler.
Dim Chan, SheetName, I, TopicList                 ' Declare variables.
Chan = DDEInitiate("Excel", "System")             ' Establish spreadsheet link.
If Err Then                                       ' If error occurs, spreadsheet
  Err = 0                                         '  isn't running. Reset error
  I = Shell("Excel", 1)                           '  and start spreadsheet.
  If Err Then Exit Sub                            ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")           ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                       ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")         ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                                 ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)            ' Establish link w/new sheet.
For I = 1 To 10                                   ' Put some values into
  DDEPoke Chan, "R1C" & I, I                      '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]" ' Make chart.
DDETerminateAll                                   ' Terminate all links.
```

DDERequest Function

Description Uses an open dynamic data exchange (DDE) channel to request an item of information from another application.

Syntax **DDERequest**(*channum*, *item*)

Remarks The **DDERequest** function uses the following arguments.

Argument	Description
channum	Channel number returned by the **DDEInitiate** function
item	String expression containing the name of a data item recognized by the other application

The actual value of *item* depends on the application and topic specified when the channel *channum* is opened. An error occurs if *channum* isn't an **Integer** corresponding to an open channel.

The **DDERequest** function returns a **Variant** containing the requested information as a string if the request was successful.

The data is requested in plain text format. Pictures or text in any other format cannot be transferred.

See Also **DDE** Function, **DDEInitiate** Function, **DDEPoke** Statement

Example This example establishes a DDE link with Microsoft Excel, places some values into cells in the first row of a new worksheet, and charts the values. **DDERequest** asks Microsoft Excel for the name of the newly created worksheet.

```
On Error Resume Next                          ' Set up error handler.
Dim Chan, SheetName, I, TopicList             ' Declare variables.
Chan = DDEInitiate("Excel", "System")         ' Establish spreadsheet link.
If Err Then                                   ' If error occurs, spreadsheet
  Err = 0                                     '  isn't running. Reset error
  I = Shell("Excel", 1)                       '  and start spreadsheet.
  If Err Then Exit Sub                        ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")       ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                   ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")     ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                             ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)        ' Establish link with new sheet.
```

```
For I = 1 To 10                                    ' Put some values into
  DDEPoke Chan, "R1C" & I, I                       '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]"       ' Make chart.
DDETerminateAll                                    ' Terminate all links.
```

DDESend Function

Description Initiates a dynamic data exchange (DDE) conversation with another application and sends an item of information to that application.

Syntax **DDESend**(*application*, *topic*, *item*, *data*)

Remarks You can use the **DDESend** function only on forms in the ControlSource property of text boxes, option groups, check boxes, and combo boxes.

The **DDESend** function uses the following arguments.

Argument	Description
application	String expression that is the name of an application that can participate in a DDE conversation. Usually, this is the name of an .EXE file (without the .EXE extension) for a Microsoft Windows–based application, such as Microsoft Excel.
topic	String expression that is the name of a topic recognized by *application*.
item	String expression that is the name of a data item recognized by *application*.
data	String containing the data sent to *application*.

If the **DDE** function successfully initiates a DDE conversation, it sends the requested information to *application*. The maximum number of DDE conversations that can be open simultaneously is determined by Microsoft Windows and your computer's memory and resources. If the conversation can't be initiated because *application* isn't running or doesn't recognize *topic*, or if the maximum number of conversations has already been reached, **DDESend** returns a **Null**.

Note The other application may be configured to ignore your DDE conversation. If so, **DDESend** returns a **Null**. Similarly, you can set Microsoft Access to ignore requests from other applications by choosing the Options command from the View menu. Then select the Ignore DDE Requests item and change its setting to Yes.

The value of *topic* depends on *application*. For applications that use documents or data files, *topic* often includes the names of those files. The value of *item* also depends on *application*. You define the topic name in the other application, not in Microsoft Access. For Microsoft Excel, *item* might be a row and column identifier, such as "R1C1", or the name of a range of cells.

The *data* argument specifies the information you want to send. It can be a literal string, such as "Report prepared by John". Or it can be an expression that includes the result of a function that creates a string, such as "Prepared on " & Date( ). If *item* refers to more than one piece of information, such as a named range in a Microsoft Excel worksheet that contains multiple cells, **DDESend** sends *data* to the first entry.

If you use the **DDESend** function in the ControlSource property setting of a text box or option group control, *data* can include the name of another control. You can't use the name of a control when you use **DDESend** in the ControlSource property setting of a check box or combo box.

In the following example, the **DDESend** function sends "Some text" to the cell at row 1, column 1 in a Microsoft Excel worksheet. You can enter this expression for a text box control in the ControlSource property in the control's property sheet.

```
=DDESend("Excel", "Sheet1", "R1C1", "Some text")
```

When you use **DDESend**, the control becomes read-only in Form view and Print Preview. Because the ControlSource property is also read-only in these views, changes to the control must be made in Design view.

Tip You can also use the Locked property to make a control read-only.

The following table illustrates how **DDESend** behaves when used with each control.

Control	Remarks
Text box	Because the text box appears blank in Form view and Print Preview, set its Visible property to No. The *data* argument can refer to another control. The following example shows how you can send the contents of Entry Field to a Microsoft Excel worksheet. =DDESend("Excel", "Sheet1", "R1C1",[Entry Field])
Option group	None of the option buttons in the option group is selected in Form view and Print Preview. You may want to make the option group (and its buttons) invisible by setting its Visible property to No. The *data* argument can refer to another control. The following example shows how you can send the contents of Entry Field to a Microsoft Excel worksheet. =DDESend("Excel", "Sheet1", "R1C1",[Entry Field])

Control	Remarks
Check box	The check box appears dimmed in Form view and Print Preview. You may want to make it invisible by setting its Visible property to No. The *data* argument must contain numeric data, such as "100". If it contains non-numeric data, such as "Product Name", **DDESend** doesn't send the information and *item* isn't changed.
Combo box	Because the combo box appears blank in Form view and Print Preview, you should set its Visible property to No. The *data* argument must contain numeric data, such as "100". If it contains non-numeric data, such as "Product Name", **DDESend** doesn't send the information and *item* isn't changed.

See Also **DDE** Function; **DDEInitiate** Function; **DDERequest** Function; Enabled, Locked Properties; Visible Property

DDETerminate Statement

Description Closes the specified dynamic data exchange (DDE) channel.

Syntax **DDETerminate** *channum*

Remarks The argument *channum* is the channel number to close. The channel must have been opened by the **DDEInitiate** function, which returns *channum*. An error occurs if *channum* isn't an open channel number.

Once a channel is closed, any subsequent DDE functions or statements performed on that channel cause an error.

See Also **DDEInitiate** Function, **DDETerminateAll** Statement

Example

This example establishes a DDE link with Microsoft Excel, places some values into cells in the first row of a new worksheet, and charts the values. **DDETerminate** terminates the DDE link with Microsoft Excel.

```
On Error Resume Next                                ' Set up error handler.
Dim Chan, SheetName, I, TopicList                   ' Declare variables.
Chan = DDEInitiate("Excel", "System")               ' Establish spreadsheet link.
If Err Then                                         ' If error occurs, spreadsheet
  Err = 0                                           '  isn't running. Reset error
  I = Shell("Excel", 1)                             '  and start spreadsheet.
  If Err Then Exit Sub                              ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")             ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                         ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")           ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                                   ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)              ' Establish link w/new sheet.
For I = 1 To 10                                     ' Put some values into
  DDEPoke Chan, "R1C" & I, I                        '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]" ' Make chart.
DDETerminateAll                                     ' Terminate all links.
```

DDETerminateAll Statement

Description

Closes all open dynamic data exchange (DDE) channels.

Syntax

DDETerminateAll

Remarks

DDETerminateAll closes all open DDE channels that have been opened by Access Basic. Using this statement is equivalent to performing a **DDETerminate** statement for each open channel number. It doesn't cause an error if no DDE channels are open. Like **DDETerminate**, **DDETerminateAll** doesn't affect active DDE link expressions in fields on forms or reports.

If you interrupt a procedure that performs DDE, you may inadvertently leave channels open. Each open channel uses system resources, and they aren't closed automatically. Therefore, if you leave channels open, you may exhaust system resources. For this reason, it is a good idea to use **DDETerminateAll** in your code or periodically from the Immediate window while debugging code that performs DDE.

See Also

DDEInitiate Function, **DDETerminate** Statement

Example This example establishes a link with Microsoft Excel, places some values in the first row of a new worksheet, and charts the values. **DDETerminateAll** terminates all active DDE links.

```
On Error Resume Next                              ' Set up error handler.
Dim Chan, SheetName, I, TopicList                 ' Declare variables.
Chan = DDEInitiate("Excel", "System")             ' Establish spreadsheet link.
If Err Then                                       ' If error occurs, spreadsheet
  Err = 0                                         '  isn't running. Reset error
  I = Shell("Excel", 1)                           '  and start spreadsheet.
  If Err Then Exit Sub                            ' If another error, exit.
  Chan = DDEInitiate("Excel", "System")           ' Establish spreadsheet link.
End If
DDEExecute Chan, "[New(1)]"                       ' Create new worksheet.
TopicList = DDERequest(Chan, "Selection")         ' Get topic list, sheet name.
SheetName = Left(TopicList, InStr(1, TopicList, "!") - 1)
DDETerminate Chan                                 ' Terminate DDE link.
Chan = DDEInitiate("Excel", SheetName)            ' Establish link w/new sheet.
For I = 1 To 10                                   ' Put some values into
  DDEPoke Chan, "R1C" & I, I                      '  first row.
Next I
DDEExecute Chan, "[Select(""R1C1:R1C10"")][New(2,2)]" ' Make chart.
DDETerminateAll                                   ' Terminate all links.
```

Debug Object

Description You can use the **Debug** object to send output to the Immediate window.

The **Debug** object has no properties and only one method: **Print**.

You can use the **Debug** object in Access Basic.

See Also **Forms** Object, **Print** Method, **Reports** Object, **Screen** Object

Example This example prints the numbers 1 through 5, their squares, and their cubes. You can enter this example in the Module window and view the results in the Immediate window.

```
Sub Print_Squares
  Dim X As Integer
  For X = 1 To 5
    Debug.Print "X ="; X, " X squared ="; X ^ 2, " X cubed ="; X ^ 3
  Next X
End Sub
```

DecimalPlaces Property

Applies To Control (text box). Table fields.

Description Determines the number of decimal places Microsoft Access uses to display numbers.

Setting Select the tool or control.

Setting	Description
Auto	(Default) Numbers appear as specified by the Format property setting.
0–15	Digits to the left of the decimal point appear as specified by the Format property setting; digits to the right of the decimal point appear with the specified number of decimal places.

Remarks Use the DecimalPlaces property to display numbers differently from the way specified by the Format property setting. For example, the Currency format displays only two decimal places ($5.35). To display Currency numbers with four decimal places (for example, $5.3523), you must use the DecimalPlaces property.

The DecimalPlaces property has no effect if the Format property is set to General Number.

Changes to default control property settings affect only controls created on the current form or report. To change the default control property settings for all new forms or reports, create a new template.

Access Basic The property settings and their values are:

Setting	Value
Auto	255
0–15	0–15

Design view: read/write. Other views: read-only.

See Also **Format**, **Format$** Functions; Format Property

Declare Statement

Description Declares references to external procedures in a dynamic-link library (DLL).

Syntax 1 **Declare Sub** *globalname* **Lib** *libname* [**Alias** *aliasname*][([*argumentlist*])]

Syntax 2 **Declare Function** *globalname* **Lib** *libname* [**Alias** *aliasname*] [([*argumentlist*])] [**As** *type*]

Remarks The **Declare** statement has these parts.

Part	Description
Sub	Indicates that the procedure doesn't return a value.
Function	Indicates that the procedure returns a value and can be used in an expression.
globalname	The name of the **Sub** or **Function** procedure called. Procedure names follow the same rules used for naming other Access Basic variables. **Function** procedure names can include a type-declaration character indicating the data type returned by the procedure. This name cannot appear as the name of any other procedure. For **Function** procedures, the data type of the procedure determines the data type it returns. If the function doesn't have a data type, you can use an **As** clause following the *argumentlist* to give it one.
Lib	Indicates that a DLL contains the procedure being declared. The **Lib** clause is required for all declarations.
libname	The name of the DLL that contains the declared procedure.
Alias	Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as an Access Basic reserved word. You can also use **Alias** when a DLL procedure has the same name as a **Global** variable or constant or any other procedure in the same scope. **Alias** is also useful if any characters in the DLL procedure name aren't allowed in Access Basic names.
aliasname	A text string that identifies the name of the procedure in the DLL.

Part	Description
argumentlist	A list of variables representing arguments that are passed to the **Sub** or **Function** procedure when it is called.
As *type*	Declares the data type of the value returned by a **Function** procedure. The argument *type* may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (variable-length only), or **Variant**.

The argument *argumentlist* has the following syntax:

[**ByVal**] *variable* [**As** *type*] [,[**ByVal**] *variable* [**As** *type*]] . . .

The following list describes the parts of *argumentlist*.

Part	Description
ByVal	Indicates that the argument is passed by value rather than by reference. The **ByVal** reserved word cannot be used with a variable of a user-defined type. When **ByVal** precedes a numeric argument *variable*, the actual argument is converted to the numeric type indicated in the **Declare** statement before being passed. When **ByVal** precedes a **String** argument *variable*, the address of the null-terminated string data is sent to the procedure. When **ByVal** is not included, a string descriptor is sent to the called DLL procedure.
variable	An Access Basic variable name. If you use a type-declaration character with *variable*, don't use the **As** clause. If there is no **As** clause, the default data type of *variable* is used.
As *type*	Declares the data type of *variable*: **Integer**, **Long**, **Single**, **Double**, **Currency**, **String**, **Variant**, **Any**, or a user-defined type. Use the **Any** data type in an **As** clause only to override type checking for that argument.

Use the **Declare** statement to declare external procedures (procedures contained in a DLL). A **Declare** statement for an external procedure can appear only in the Declarations section of a module. DLL procedures declared in any module are available to all procedures in all modules.

Empty parentheses indicate that the **Sub** or **Function** procedure has no arguments and that arguments should be checked to ensure that none are passed. In the following example, `First` takes no arguments. If you use arguments in a call to `First`, an error occurs:

```
Declare Sub First Lib "MyLib" ()
```

When an argument list appears, the number and type of arguments are checked each time the procedure is called. In the following example, `First` takes one **Long** argument:

```
Declare Sub First Lib "MyLib" (X&)
```

Note You can't have fixed-length strings in the parameter list of a **Declare** statement because only variable-length strings can be passed to procedures. Fixed-length strings can appear as procedure arguments, but they are converted to variable-length strings before being passed.

See Also **Call** Statement

Example In this example, the **Declare** statement declares a reference to an external function (`GetProfileString`) in the Windows Kernel DLL. Before attempting to run this example, make sure that the entire **Declare** statement appears on a single line in the Declarations section.

```
Declare Function GetProfileString Lib "Kernel" (ByVal SName$, ByVal
➥ KName$,ByVal Def$, ByVal Ret$, ByVal Size%) As Integer
Sub Declare_Demo ()
SName$ = "Intl"                                    ' WIN.INI section name.
KName$ = "sCountry"                                ' WIN.INI country code.
Ret$ = String$(255, 0)                             ' Initialize return string.
' Call Windows Kernel DLL.
Success = GetProfileString(SName$, KName$, "", Ret$, Len(Ret$))
If Success Then                                    ' Evaluate results.
  Msg = "'" & KName$ & "' = " & Ret$
Else
  Msg = "There is no country code in your WIN.INI file."
End If
MsgBox Msg                                         ' Display message.
End Sub
```

Default Property

Applies To Control (command button) on a form.

Description Determines whether a command button is the default button on a form.

Setting The Default property settings are:

Setting	Description
Yes	The command button is the default button.
No	(Default) The command button isn't the default button.

Remarks Only one command button on a form can be the default button. When the Default property is set to Yes for one command button, it is automatically set to No for all other command buttons on the form. When the command button's Default property setting is Yes and the form is the active form, the user can choose the command button by clicking it or by pressing ENTER when no other command button has the focus.

Tip For a form that supports irreversible operations, such as deletions, it's a good idea to make the Cancel button the default button. To do this, set both the Default property and the Cancel property to Yes.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only

DefaultEditing Property

See AllowEditing, DefaultEditing Properties.

DefaultValue Property

Applies To Table fields. Controls (check box*, combo box, list box, option button*, option group, text box, toggle button*) on a form.

* Except when the control is in an option group.

Description Specifies the default value for a field or control.

Setting Enter text or an expression. The maximum length is 255 characters.

Remarks The default value is automatically entered in a field or control when a new record is created. For example, in an address table you might set the default for the City field to New York. When users add records to the table, they can either accept this value or enter the name of a different city.

If you create a control by dragging a field from the field list, the field's DefaultValue is copied to the control's DefaultValue.

The following example shows how you can use one control to provide the default value for another. Notice, however, that the two controls cannot be on the same form. You can enter this expression as the setting for the DefaultValue property for the control on Form1.

=**Forms**![Form2]![My Control].DefaultValue

Note The DefaultValue property doesn't apply to fields whose data type is Counter or OLE Object.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only. Table fields aren't available.

DefaultView, ViewsAllowed Properties

Apply To Forms.

Description

- DefaultView—specifies the opening view of a form.
- ViewsAllowed—determines whether users can switch between Datasheet view and Form view by choosing the Form or Datasheet command from the View menu.

Setting The DefaultView property settings are:

Setting	Description
Single Form	Displays one record at a time
Continuous Forms	(Default) Displays multiple records, each in its own copy of the detail section of the form
Datasheet	Displays the form fields arranged in rows and columns like a spreadsheet

The ViewsAllowed property settings are:

Setting	Description
Both	(Default) Users can switch between Form view and Datasheet view.
Form	Users can't switch to Datasheet view.
Datasheet	Users can't switch to Form view.

Design view is always available.

Remarks

Users can view data in Form view or Datasheet view. In Form view, a form displays as a single form or as continuous multiple forms. The DefaultView and ViewsAllowed properties determine how users see a form and whether they can see it in different views.

The combination of these properties creates the following conditions.

ViewsAllowed	DefaultView	Description
Both	Single, Continuous Forms, or Datasheet	Users can switch between Form view and Datasheet view.
Form	Single or Continuous Forms	Users can't switch from Form view to Datasheet view.
Form	Datasheet	Users can switch to Form view but not back to Datasheet view. (Not recommended.)
Datasheet	Single or Continuous Forms	Users can switch to Datasheet view but not back to Form view. (Not recommended.)
Datasheet	Datasheet	Users can't switch from Datasheet view to Form view.

Access Basic

The DefaultView property settings and their values are:

Setting	Value
Single Form	0
Continuous Forms	1
Datasheet	2

Design view: read/write. Other views: read-only.

The ViewsAllowed property settings and their values are:

Setting	Value
Both	0
Form	1
Datasheet	2

Design view: read/write. Other views: read-only.

Def*type* Statements

Description Used in the Declarations section of modules to set the default data type for variables and **Function** procedures.

Syntax

DefInt *letterrange* [,*letterrange*] . . .

DefLng *letterrange* [,*letterrange*] . . .

DefSng *letterrange* [,*letterrange*] . . .

DefDbl *letterrange* [,*letterrange*] . . .

DefCur *letterrange* [,*letterrange*] . . .

DefStr *letterrange* [,*letterrange*] . . .

DefVar *letterrange* [,*letterrange*] . . .

Remarks The argument *letterrange* has the following form:

letter1 [-*letter2*]

The arguments *letter1* and *letter2* specify the range of variables for which a default data type will be set. Each argument represents the first letter of variable names and can be any uppercase or lowercase letter of the alphabet. For instance, a letter range of A–D sets the default data type for all variables that begin with A, B, C, or D. The case of the letters in *letterrange* isn't significant.

The reserved word you use (*type* in **Def***type*) determines which of the following data types will be the default setting of variables in *letterrange*: **Integer(DefInt)**, **Long(DefLng)**, **Single(DefSng)**, **Double(DefDbl)**, **Currency(DefCur)**, **String(DefStr)**, or **Variant (DefVar)**. For example, in the following program fragment, `Message` is a string variable:

```
DefStr A-Q
...
Message = "Out of stack space."
```

A **Def***type* statement affects only the module in which it is used. For example, a **DefInt** statement in one module affects the default data type only of variables declared in that module; the default data type of variables in other modules is unaffected. If not explicitly declared with a **Def***type* statement, the default data type for all variables and all **Function** procedures in a module is **Variant**.

When a letter range is specified, it usually defines the data type for variables that begin with letters in the first 128 characters of the ANSI character set. However, when you specify the letter range A–Z, you set the default to the specified data type for all variables, including any that begin with international characters from the extended part of the ANSI character set (128–255).

Once the range A–Z has been specified, you cannot further redefine any subranges of variables using **Def***type* statements. In fact, once a range has been specified, if you include a previously defined letter in another **Def***type* statement, an error occurs and a message is

displayed. However, you can explicitly assign the data type of any variable, defined or not, using a **Dim** statement or a type-declaration character. For example, you can use the following code in the Declarations section of a form or module to define a variable as a **Double** even though the default data type is **Integer**:

```
DefInt A-Z
Dim TaxRate As Double
```

You can use the following statement in a procedure to explicitly assign the data type of `Pi`:

```
Pi! = 3.141593
```

A type-declaration character always takes precedence over a **Def***type* statement. **Def***type* statements don't affect elements of user-defined types.

See Also **Let** Statement

Example In this example, the **Def***type* statements set default data types of variables in two ranges. Variables beginning with the letters A–K will be **Integer**; variables beginning with L–Z will be **String**.

```
DefInt A-K
DefStr L-Z
```

Delete Method

Description Deletes the current record in a specified **Table** or **Dynaset**.

Syntax *object*.**Delete**

Remarks The **Delete** method uses the following argument.

Argument	Description
object	Name of an open **Table** or **Dynaset** object containing the record you want to delete

There must be a current record in *object* before you use **Delete**; otherwise, an error occurs. To make a record current, move to the desired record using one of the Move methods, one of the Find methods (**Dynasets** only), or the **Seek** method (**Tables** only).

In a **Table**, **Delete** removes the current record. In a **Dynaset**, it sets the current record to **Null** and removes the current record from its underlying **Tables**. Although you can't edit or use it, the deleted record remains current.

Subsequent references to a deleted record in a **Table** are invalid and produce an error.

What was the next record remains the next record, and the previous record remains the previous record. Once you move to another record, however, you can't make the deleted record current again.

You can undelete a record if you use transactions and the **Rollback** statement.

See Also **AddNew** Method, **BeginTrans** Statement, **Edit** Method, **Rollback** Statement

Example This example adds a record to the Customers table in the NWIND.MDB database and then uses **Delete** to remove it.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Customers")      ' Open Table.
MyTable.Index = "PrimaryKey"                   ' Set index.
MyTable.AddNew                                 ' Prepare new record.
MyTable("Customer ID") = "FINNF"               ' Set record key.
MyTable("Company Name") = "Finnegan's Foods"   ' Set company name.
MyTable.Update                                 ' Save changes.
...
MyTable.Seek "=", "FINNF"                      ' Make new record current.
MyTable.Delete                                 ' Delete record.
MyTable.Close                                  ' Close Table.
```

DeleteQueryDef Method

Description Deletes a specified **QueryDef** (an object containing the SQL statement that describes a query) from a database.

Syntax *database*.**DeleteQueryDef**(*name*)

Remarks The **DeleteQueryDef** method uses the following arguments.

Argument	Description
database	Name of an open database object
name	String expression that is the name of an existing **QueryDef**

Once you use **DeleteQueryDef**, subsequent references to *name* are invalid and produce an error.

If the **QueryDef** named by *name* doesn't exist or is open, or if *database* doesn't refer to an open database, an error occurs.

To create a new **QueryDef**, use the **CreateQueryDef** method. To change an existing **QueryDef**, use the **OpenQueryDef** method and change the appropriate properties.

See Also **CreateQueryDef** Method, **OpenQueryDef** Method

Example This example creates and then deletes the `All Cust` **QueryDef**. It also creates `ListSet`, a **Snapshot** that contains records for all customers in the Customers table of the NWIND.MDB database.

```
Dim MyDB As Database, MyQuery As QueryDef, ListSet As Snapshot
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("All Cust")        ' Create QueryDef.
MyQuery.SQL = "SELECT * FROM Customers;"             ' Set SQL property.
Set ListSet = MyDB.CreateSnapshot("All Cust")        ' Create Snapshot.
MyQuery.Close                                        ' Close QueryDef.
MyDB.DeleteQueryDef("All Cust")                      ' Delete QueryDef.
```

Derived Math Functions

Description The following is a list of nonintrinsic mathematical functions that can be derived from the intrinsic math functions provided with Access Basic.

Function	Access Basic equivalent
Secant	`Sec(X) = 1 / Cos(X)`
Cosecant	`Cosec(X) = 1 / Sin(X)`
Cotangent	`Cotan((X) = 1 / Tan(X)`
Inverse Sine	`Arcsin(X) = Atn(X / Sqr(-X * X + 1))`
Inverse Cosine	`Arccos(X) = Atn(X / Sqr(-X * X + 1)) + 1.5708`
Inverse Secant	`Arcsec(X) = Atn(X / Sqr(X * X - 1)) + Sgn(Sgn(X) -1) * 1.5708`
Inverse Cosecant	`Arccosec(X) = Atn(X/Sqr(X * X - 1)) + (Sgn(X) - 1) * 1.5708`
Inverse Cotangent	`Arccotan(X) = Atn(X) + 1.5708`
Hyperbolic Sine	`HSin(X) = (Exp(X) - Exp(-X)) / 2`
Hyperbolic Cosine	`HCos(X) = (Exp(X) + Exp(-X)) / 2`
Hyperbolic Tangent	`HTan(X) = (Exp(X) - Exp(-X)) / (Exp(X) + Exp(-X))`
Hyperbolic Secant	`HSec(X) = 2 / (Exp(X) + Exp(-X))`
Hyperbolic Cosecant	`HCosec(X) = 2 / (Exp(X) - Exp(-X))`

Function	Access Basic equivalent
Hyperbolic Cotangent	`HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))`
Inverse Hyperbolic Sine	`HArcsin(X) = Log(X + Sqr(X * X + 1))`
Inverse Hyperbolic Cosine	`HArccos(X) = Log(X + Sqr(X * X - 1))`
Inverse Hyperbolic Tangent	`HArctan(X) = Log((1 + X) / (1 - X)) / 2`
Inverse Hyperbolic Secant	`HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)`
Inverse Hyperbolic Cosecant	`HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) +1) / X)`
Inverse Hyperbolic Cotangent	`HArccotan(X) = Log((X + 1) / (X - 1)) / 2`
Logarithm	`LogN(X) = Log(X) / Log(N)`

Description Property

Applies To Table fields.

Description Specifies the text that describes a table and its fields.

Setting Enter the table description in the property sheet; enter the field description in the table's Design view. The maximum length is 255 characters.

Remarks Use the Description property to provide information about the table and its fields.

If you create controls by dragging a field from the field list, Microsoft Access copies the field's Description to the control's StatusBarText property.

Note When you attach a table, Microsoft Access displays a description of the attached table in the property sheet.

Access Basic This property isn't available from Access Basic.

See Also Caption Property

DFirst, DLast Functions

Description Return a field value from the first and last record in a specified set of records (domain).

Syntax **DFirst**(*expr*, *domain* [, *criteria*])

DLast(*expr*, *domain* [, *criteria*])

Remarks The **DFirst** and **DLast** functions use the following arguments.

Argument	Description
expr	String expression identifying the field that contains the data you want to return, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DFirst** or **DLast** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, *expr* is evaluated against the entire domain.

DFirst and **DLast** are similar to **DLookup**. These functions return a field value from the record that satisfies *criteria*. **DFirst** and **DLast** return values from the first and last occurrence according to the order of records in *domain*. If *domain* is an indexed table, the order follows the current index. Otherwise, the order follows the actual order of the records.

If more than one record satisfies *criteria*, **DFirst** returns the field value from the first occurrence.

You can use **DFirst** and **DLast** to determine the limits of a domain. For example, the fields returned by **DFirst** and **DLast** might contain order dates that you could use in a report title to indicate the range for orders.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table in the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DFirst("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DLast("[Freight]", "Orders", "[Ship Country] = 'UK'")
Z = DLast("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use these functions in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DFirst("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If no record satisfies *criteria*, or if *domain* contains no records, **DFirst** and **DLast** return a **Null**.

If any records in *domain* are being edited when you use **DFirst** or **DLast**, the function uses the most recently saved values.

See Also **DLookup** Function; **FindFirst, FindLast, FindNext, FindPrevious** Methods; **First, Last** Functions

Example This example returns cost information from the Freight field of the first and last record for orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK`.

```
X = DFirst("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DLast("[Freight]", "Orders", "[Ship Country] = 'UK'")
```

Dim Statement

Description Used at the module level and at the procedure level to declare variables and allocate storage space.

Syntax **Dim** *variablename*[([*subscripts*])][**As** *type*][, *variablename*[([*subscripts*])][**As** *type*]] . . .

Remarks The **Dim** statement uses these arguments.

Argument	Description
variablename	Name of a variable.
subscripts	Dimensions of an array variable. You can declare multiple dimensions. The syntax of *subscripts* is described below.
As *type*	Reserved word used to declare the data type of a variable. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Variant**, a user-defined type, or an object data type (but no arrays of objects). Use a separate **As** *type* clause for each variable being defined.

The argument *subscripts* in the **Dim** statement has the following syntax:

[*lower* **To**]*upper*[,[*lower* **To**] *upper*] . . .

The **To** reserved word provides a way to indicate both the lower and upper bounds of an array variable's subscripts. The following statements are equivalent if there is no **Option Base** statement:

```
Dim A(8,3)
Dim A(0 To 8, 0 To 3)
Dim A(8, 0 To 3)
```

Array subscripts can be negative. **To** can be used to specify any range of subscripts between –32,768 and 32,767, inclusive:

```
Dim A(-4 To 10)
Dim B(-99 To -5, -3 To 0)
```

The maximum number of array dimensions allowed in a **Dim** statement is 60.

You can calculate the amount of memory used by a numeric array by multiplying the number of elements in the array times the number of bytes required by the data type of the array. For example, an **Integer** array, which requires 2 bytes per element, can contain twice as many elements as a **Long** array, which requires 4 bytes per element. An **Integer** array can contain four times as many elements as a **Double** array, which requires 8 bytes per element. **String** arrays are limited to less than 64K bytes because some small storage overhead is required for each array.

If you use a subscript that is greater than the specified maximum or smaller than the specified minimum, an error occurs and a message is displayed, unless it is trapped in error-handling code. An error also occurs if the size of the array (in terms of bytes of memory used) exceeds the allowable limits described above.

Use **Dim** in the Declarations section of a module to declare variables that are available to all procedures throughout that module.

Use **Dim** in a **Sub** or **Function** procedure to declare variables that are local to that procedure. It is generally accepted programming practice to put **Dim** statements at the beginning of the procedure.

Use a **Dim** statement in the Declarations section or in **Sub** or **Function** procedures to declare the data type of a variable. For example, the following statement declares the variable as an **Integer**.

```
Dim NumberOfEmployees As Integer
```

You can also use the **Dim** statement with empty parentheses to declare dynamic arrays. After declaring a dynamic array, use the **ReDim** statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size has already been declared, an error occurs.

Variables are initialized at compile time. Numeric variables are initialized to 0, **Variant** variables to Empty. Variable-length strings are initialized as zero-length strings, and fixed-length strings are filled with zeros. The fields of user-defined type variables are initialized as if they were separate variables.

See Also **Global** Statement, **Option Base** Statement, **ReDim** Statement, **Static** Statement, **Type** Statement

Example This example uses the **Dim** statement to declare two variables in a module. The first variable is a dynamic-array variable; the second is an integer variable.

```
Dim TestArray() As Integer                ' Declare dynamic-array variable.
Dim Size As Integer                       ' Declare integer variable.
```

Dir, Dir$ Functions

Description Return a file name that matches a specified pattern.

Syntax **Dir**[$] [(*filespec*)]

Remarks **Dir** returns a **Variant**; **Dir$** returns a **String**. When **Dir**[$] is used in anything except Access Basic code in a module, the empty parentheses must be included when there is no *filespec* argument.

The argument *filespec* is a string expression that specifies a path or file name. The path and file name can include a drive specification and any valid wildcard characters.

You must specify *filespec* the first time you call the function, or an error occurs. **Dir**[$] returns the first file name that matches *filespec*. To get any additional file names that match *filespec*, call **Dir**[$] again with no argument. When no more file names match, **Dir**[$] returns a zero-length **String** or **Variant** of **VarType** 8. Once a zero-length **String** or **Variant** has been returned, you must again use a *filespec* argument in subsequent calls or an error occurs.

Note You can change to a new *filespec* without retrieving all of the file names that match the current *filespec*.

Tip Because file names are retrieved in no particular order, you may want to store returned file names in an array and then sort the array.

See Also **ChDir** Statement; **CurDir, CurDir$** Functions

Example This example uses **Dir** to display the names of files that match a file specification the user provides.

```
Msg = "Enter a file specification."              ' Create message.
Filespec = InputBox(Msg)                         ' Get file name.
Match = Dir(Filespec)                            ' Find first match.
If Len(Match) > 0 Then                           ' If match found.
  Do
    MsgBox Match                                 ' Display matching file name.
    Match = Dir                                  ' Find next match.
  Loop Until Len(Match) = 0                      ' Continue.
  Msg = "There are no more matching files."
Else
  Msg = "The file you specified could not be found."
End If
MsgBox Msg                                       ' Display message.
```

DisplayWhen Property

Applies To Form sections (detail section, form footer, form header). Tools and controls (all except page breaks) on a form.

Description Determines whether an object is displayed or printed.

Setting The DisplayWhen property settings are:

Setting	Description
Always	(Default) Object appears in Form view and when printed.
Print Only	Object is hidden in Form view but appears when printed.
Screen Only	Object appears in Form view but not when printed.

Use the DisplayWhen property to specify which objects you want displayed on screen and in print. For example, you might have a command button or instructions on a form that you don't want printed. Or you might have form header and form footer sections that you don't want displayed on screen but that you do want printed.

Changes to default control properties are in effect only for the current form. To change default values for all new forms, create a new template.

Note For reports, use the OnFormat property to specify a macro that sets the Visible property of controls you don't want printed.

Access Basic

The property settings and their values are:

Setting	Value
Always	0
Print Only	1
Screen Only	2

Design view: read/write. Other views: read-only.

See Also Enabled, Locked Properties; SetValue Action

DLast Function

See **DFirst**, **DLast** Functions

DLookup Function

Description Returns a field value in a specified set of records (domain).

Syntax **DLookup**(*expr*, *domain* [, *criteria*])

Remarks The **DLookup** function uses the following arguments.

Argument	Description
expr	String expression identifying the field that contains the data you want to return, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).

Argument	Description
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DLookup** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DLookup** evaluates *expr* against the entire domain.

You can use **DLookup** to return a field value based on the value of another field. In the following example from the Employees table in the NWIND.MDB database, **DLookup** uses the Employee ID field to return the corresponding last name.

```
X = DLookup("[Last Name]", "Employees", "[Employee ID] = 24")
```

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table of the NWIND.MDB database, `Ship Postal Code` is the name of a field, and `98008` is a string literal.

```
X = DLookup("[Ship Via]", "Orders", "[Ship Postal Code] = '94117'")
Y = DLookup("[Ship Via]", "Orders", "[Ship Via] = 1")
```

If you use **DLookup** in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DLookup("[Freight]", "Orders", "[Region] = Form.[Region]")
```

Even if more than one record satisfies *criteria*, **DLookup** returns only one field. If no record satisfies *criteria*, or if *domain* contains no records, **DLookup** returns a **Null**.

If any records in *domain* are being edited when you use **DLookup**, the function uses the most recently saved values.

See Also **DFirst**, **DLast** Functions

Example This example returns name information from the Company Name field of the record satisfying the criteria. The domain is the Shippers table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to the record in which `Shipper ID` equals `1`.

```
X = DLookup("[Company Name]", "Shippers", "[Shipper ID] = 1")
```

The following example from the Shippers table in the NWIND.MDB database uses the form control `MyControl` to provide the lookup criteria.

```
X = DLookup("[Company Name]", "Shippers", "[Shipper ID] =
➥ Forms![MyForm]![MyControl]")
```

The next example uses a replaceable parameter, `SearchID`, to perform a lookup.

```
SearchID = 1
X = DLookup("[Company Name]", "Shippers", "[Shipper ID] = " & SearchID)
```

DMax Function

See **DMin, DMax** Functions

DMin, DMax Functions

Description Return the minimum and maximum of a set of values in a specified set of records (domain).

Syntax **DMin**(*expr*, *domain* [, *criteria*])

DMax(*expr*, *domain* [, *criteria*])

Remarks The **DMin** and **DMax** functions use the following arguments.

Argument	Description
expr	String expression identifying the field that contains the data you want to evaluate, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).

Argument	Description
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DMin** and **DMax** are performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DMin** and **DMax** evaluate *expr* against the entire domain.

You can use **DMin** and **DMax** to return the lowest and highest value. If *expr* identifies numeric data, these functions return numeric values. If *expr* identifies non-numeric data, they return the string that is first or last alphabetically.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table of the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DMin("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DMax("[Freight]", "Orders", "[Ship Country] = 'UK'")
Z = DMax("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use these functions in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DMin("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If any records in *domain* are being edited when you use **DMin** or **DMax**, the function uses the most recently saved values.

See Also **DFirst**, **DLast** Functions; **Min**, **Max** Functions

Example This example returns the lowest and highest value from the Freight field for orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK`.

```
X = DMin("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DMax("[Freight]", "Orders", "[Ship Country] = 'UK'")
```

The next example returns the same values using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DMin("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
Y = DMax("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
```

Do...Loop Statement

Description Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax 1

Do [{**While** | **Until**} *condition*]
  [*statementblock*]
  [**Exit Do**]
  [*statementblock*]
Loop

Syntax 2

Do
  [*statementblock*]
  [**Exit Do**]
  [*statementblock*]
Loop [{**While** | **Until**} *condition*]

Remarks The argument *condition* is a numeric or string expression that evaluates true (nonzero) or false (0 or **Null**). The program lines between the **Do** and **Loop** statements are repeated while or until *condition* is true.

An **Exit Do** statement, which may be used only inside a **Do...Loop**, provides an alternate way to exit a **Do...Loop** control structure. Any number of **Exit Do** statements may be placed anywhere within the **Do...Loop**. Often used with the evaluation of some condition (for example, **If...Then**), **Exit Do** transfers control to the statement immediately following the **Loop** statement. When **Do...Loop** statements are nested, control is transferred to the **Do...Loop** that is one nested level above the loop in which the **Exit Do** occurs.

Note The **Option Compare** statement affects string comparison.

See Also **While...Wend** Statement

Example This example sets up an infinite **Do...Loop** that can be exited only if the user enters a number in a range.

```
Do
  Answer = InputBox("Enter a number greater than 1 and less than 9.")
  If Answer > 1 And Answer < 9 Then      ' Check range.
    Exit Do                             ' Exit Do Loop.
  End If
Loop
```

DoCmd Statement

Description Executes a Microsoft Access action.

Syntax **DoCmd** *actionname* [*actionargs*]

Remarks The **DoCmd** statement uses the following arguments.

Argument	Description
actionname	Name of the Microsoft Access action you want to execute.
actionargs	**Variant** expression that contains any arguments required by the action. Separate multiple arguments with commas.

You use **DoCmd** to execute Microsoft Access actions from Access Basic. Actions perform important tasks, such as closing windows, opening forms, and changing the mouse pointer to an hourglass.

Some actions have optional arguments. If you omit them, the arguments assume the default values for the particular action. For example, the OpenForm action uses seven arguments; only the first argument is required. The following example shows how you can open the Orders form in the NWIND.MDB database; only sales representatives are included.

```
DoCmd OpenForm "Employees", , ,"[Title] = 'Sales Representative'"
```

From Access Basic, you can execute all of the actions except:

- AddMenu
- MsgBox
- RunApp
- RunCode
- SendKeys
- SetValue
- StopAllMacros
- StopMacro

For more information about using **DoCmd** with an action, consult the Access Basic section of the relevant action topic.

See Also **Shell** Function

Example

This example uses the **DoCmd** statement several times to open the Orders table of the NWIND.MDB database in Datasheet view, maximize the window, and print the current order. Microsoft Access automatically declares these constants: `A_FORMBAR`, `A_EDITMENU`, `A_HIGH`, `A_SELECTION`, and `A_SELECTRECORD`.

```
Sub Open_Orders
  Const MB_ICONQUESTION = 32
  Const MB_ICONEXCLAMATION = 48
  Const YES = 6
  Const YES_NO = 4
  DoCmd OpenTable "Orders", A_NORMAL                          ' Open table.
  DoCmd Maximize                                              ' Maximize window.
  DoCmd DoMenuItem A_FORMBAR, A_EDITMENU, A_SELECTRECORD' Select record.
  If MsgBox("Print selected order?", MB_ICONQUESTION + YES_NO, "Print Order")
  ➥ = YES Then
    DoCmd Print A_SELECTION, , , A_HIGH, 1, False             ' Print order.
  End If
  MsgBox "Click OK to close the Orders table.", MB_ICONEXCLAMATION
  DoCmd Close                                                 ' Close window.
End Sub
```

The next example shows how you can invoke the File Save command on the current menu bar.

```
DoCmd DoMenuItem A_FORMBAR, A_FILE, A_SAVEFORM
```

DoEvents Function, DoEvents Statement

Description

Cause Access Basic to yield execution so the operating environment can process events.

Syntax

Function: **DoEvents( )**

Statement: **DoEvents**

Remarks

The **DoEvents** function, which always returns 0, can be used only in Access Basic code. However, it is recommended that you use the **DoEvents** statement instead.

DoEvents passes control to the operating environment. Control is not returned until the environment has finished processing the events in its queue and all keys in the **SendKeys** queue have been sent.

Use **DoEvents** to interrupt processing so the operating environment can respond normally to other events, such as keyboard input and mouse clicks. If parts of your code take up too much processor time, use **DoEvents** to relinquish control to the operating environment.

Caution Make sure the procedure that has given up control with **DoEvents** is not executed again by a different part of your program before the first **DoEvents** call returns; this could cause unpredictable results. In addition, do not use **DoEvents** if other applications could possibly interact with your procedure in unforeseen ways during the time you have yielded control.

Example In this example, **DoEvents** yields to the operating environment only in the second loop, allowing keystrokes.

```
MsgBox "Try pressing a key."
For I = 1 To 25000                          ' Start first loop.
  N = Int(5000 * Rnd + 1)                   ' Generate random number.
Next I                                      ' Increment loop counter.
Beep                                        ' First loop is done.
MsgBox "Try again."
For I = 1 To 25000                          ' Start second loop.
  N = Int(5000 * Rnd + 1)                   ' Generate random number.
  If I Mod 1000 = 0 Then DoEvents           ' Yield to other events.
Next I                                      ' Increment loop counter.
Beep : Beep                                 ' Second loop is done.
```

DoEvents Statement

See **DoEvents** Function, **DoEvents** Statement

DoMenuItem Action

Description Carries out a Microsoft Access menu command.

Action argument	Description
Menu Bar	The name of the menu bar that contains the command you want to carry out. This argument's drop-down list in the Action Arguments section of the Macro window shows the different views in which menu bars appear. Required argument.
Menu Name	The name of the menu that contains the command you want to carry out. The drop-down list shows all menus on the menu bar that you selected with the Menu Bar argument. Required argument.

Action argument	Description
Command	The command you want to carry out. The drop-down list shows all commands in the menu that you selected with the Menu Name argument. Required argument.
Subcommand	The subcommand you want to carry out. This applies only if the desired command has a submenu.

Remarks

You can use this action to do the following:

- Carry out any Microsoft Access command from a macro. You must select a command that is appropriate for the view you're in when the macro carries out the command. For example, a macro can't carry out any of the Records menu commands if the active object is in Design view.
- Add a Microsoft Access command to a menu on a form's custom menu bar, which replaces the standard menu bar. For each command you want to add, enter a DoMenuItem action in the macro group for that menu.

The DoMenuItem action has the same effect as choosing the specified command from a Microsoft Access menu. If choosing a command makes a dialog box appear, it will also appear when you run the DoMenuItem action. You can use the SendKeys action to enter the appropriate information in the dialog box.

Tip Because custom menu bars are displayed in Form view, you'll usually select Form for the Menu Bar argument when you put Microsoft Access commands into a menu's macro group.

See Also

AddMenu Action, SendKeys Action

Access Basic

Syntax

DoCmd DoMenuItem *menubar*, *menuname*, *command* [, *subcommand*]

Argument	Description
menubar	Use the intrinsic constant `A_FORMBAR` for the menu bar in Form view. For other views, go to the Macro window, and use the number of the view in the Menu Bar argument drop-down list (count down the list, starting from 0).

Argument	Description
menuname	You can use one of the following intrinsic constants: A_FILE A_EDITMENU A_RECORDSMENU For other menus, go to the Macro window, and use the number of the menu in the Menu Name argument drop-down list (count down the list, starting from 0).
command	You can use one of the following intrinsic constants: A_NEW A_SAVEFORM A_SAVEFORMAS A_SAVERECORD A_UNDO A_UNDOFIELD A_CUT A_COPY A_PASTE A_DELETE A_SELECTRECORD A_SELECTALLRECORDS A_OBJECT A_REFRESH For other commands, go to the Macro window, and use the number of the command in the Command argument drop-down list (count down the list, starting from 0).
subcommand	You can use one of the following intrinsic constants: A_OBJECTVERB A_OBJECTUPDATE The A_OBJECTVERB constant represents the first command on the submenu for the Object command on the Edit menu. The type of object determines the subcommand. For example, this subcommand is Change to Picture for an editable Paintbrush object. For other subcommands, go to the Macro window, and use the number of the subcommand in the Subcommand argument drop-down list (count down the list, starting from 0).

Remarks

The selections that appear in the drop-down lists for the Menu Name, Command, and Subcommand arguments in the Macro window depend on what you have selected for the previous arguments. You must use numbers or intrinsic constants that are appropriate for each *menubar*, *menuname*, *command*, and *subcommand* argument.

If you leave the *subcommand* argument blank, don't use a comma following the *command* argument.

Example

This example uses the DoMenuItem action to run the Paste command from the Edit menu in Form view.

```
DoCmd DoMenuItem A_FORMBAR, A_EDITMENU, A_PASTE
```

The next example runs the Tile command from the Window menu in Form view.

```
DoCmd DoMenuItem A_FORMBAR, 4, 0
```

Double Data Type

Description **Double** (double-precision floating-point) variables are stored as 64-bit numbers (8 bytes) ranging in value from –1.79769313486232E308 to –4.94065645841247E–324 for negative values, from 4.94065645841247E–324 to 1.79769313486232E308 for positive values, and 0. The type-declaration character for a **Double** is # (ANSI character 35).

Each floating-point value consists of three parts: the sign, the exponent, and the mantissa. In a double-precision number, the sign takes 1 bit, the exponent takes 11 bits, and the mantissa uses the remaining 52 bits and an additional implied bit.

See Also Access Basic Data Types, **Single** Data Type

DrawMode Property

Applies To Reports.

Description Specifies how the pen (the color used in drawing) interacts with existing colors when **Line**, **Circle**, and **PSet** methods are used to draw on a report when printing.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting The DrawMode property settings are:

Setting	Description
1	Black Pen.
2	Inverse of setting 15.
3	Combination of the colors common to the background color and the inverse of the pen.
4	Inverse of setting 13.
5	Combination of the colors common to both the pen and the inverse of the display.
6	Inverse of the display color.
7	Combination of the colors in the pen and in the display color, but not in both.
8	Inverse of setting 9.
9	Combination of the colors common to both the pen and the display.
10	Inverse of setting 7.
11	No operation—output remains unchanged. In effect, this setting turns drawing off.
12	Combination of the display color and the inverse of the pen color.
13	(Default) Color specified by the ForeColor property.
14	Combination of the pen color and the inverse of the display color.
15	Combination of the pen color and the display color (Merge Pen).
16	White Pen.

Usage Design view: not available. Other views: read/write.

The value of this property is an **Integer** data type.

Remarks Use this property to produce visual effects when drawing on a report. Microsoft Access compares each pixel in the draw pattern to the corresponding pixel in the existing background to determine how the drawing appears. For example, setting 7 uses the **Xor** operator to combine a draw-pattern pixel with a background pixel.

The effect of a DrawMode setting depends on the way the color of a line drawn at run time combines with the background color and with colors already drawn. Settings 1, 6, 7, 11, 13, and 16 yield the most predictable results.

See Also BackColor Property, BackStyle Property, DrawStyle Property, DrawWidth Property, FillColor Property, ForeColor Property, OnFormat Property, OnPrint Property, **QBColor** Function, **RGB** Function

DrawStyle Property

Applies To Reports.

Description Specifies the line style for the **Line** and **Circle** methods.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting The DrawStyle property settings are:

Setting	Description
0	(Default) Solid
1	Dash
2	Dot
3	Dash-dot
4	Dash-dot-dot
5	Invisible
6	Inside solid

Usage Design view: not available. Other views: read/write.

The value of this property is an **Integer** data type.

Remarks If the DrawWidth property setting is greater than 1, DrawStyle settings 1 through 4 produce a solid line (the DrawStyle property value isn't changed). Setting DrawWidth to 1 allows DrawStyle to produce the results described in the above table.

See Also BackColor Property, BackStyle Property, DrawMode Property, FillColor Property, ForeColor Property, OnFormat Property, OnPrint Property, **PSet** Method, **QBColor** Function, **RGB** Function

DrawWidth Property

Applies To Reports.

Description Specifies a line width for the **Line**, **Circle**, and **PSet** methods.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting You can set DrawWidth to a value of 1 through 32,767. This value represents the width of the line in pixels. The default is 1, or 1 pixel wide.

Usage Design view: not available. Other views: read/write.

The value of this property is an **Integer** data type.

Remarks Increase the value of this property to increase the width of the line. If DrawWidth is greater than 1, the DrawStyle property settings 1 through 4 produce a solid line (the DrawStyle property value isn't changed). Setting DrawWidth to 1 allows DrawStyle to produce the results shown in the DrawStyle property table.

See Also BackColor Property, BackStyle Property, DrawMode Property, FillColor Property, ForeColor Property, OnFormat Property, OnPrint Property, **QBColor** Function, **RGB** Function

DStDev, DStDevP Functions

Description Return estimates of the standard deviation for a population and a population sample represented as a set of values in a specified set of records (domain).

Syntax **DStDev**(*expr*, *domain* [, *criteria*])

DStDevP(*expr*, *domain* [, *criteria*])

Remarks **DStDevP** evaluates a population, and **DStDev** evaluates a population sample.

The **DStDev** and **DStDevP** functions use the following arguments.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to evaluate, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DStDev** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DStDev** evaluates *expr* against the entire domain.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table in the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DStDev("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DStDevP("[Freight]", "Orders", "[Ship Country] = 'UK'")
Z = DStDevP("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use these functions in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DStDev("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If *domain* refers to fewer than two records, or if fewer than two records satisfy *criteria*, **DStDev** and **DStDevP** return a **Null** (which indicates that a standard deviation can't be calculated).

If any records in *domain* are being edited when you use **DStDev** or **DStDevP**, the function uses the most recently saved values.

See Also **StDev**, **StDevP** Functions

Example

This example returns estimates of the standard deviation for a population and a population sample for orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK`.

```
X = DStDev("[Freight]", "Orders", "[Ship Country] = 'UK'")  ' Sample.
Y = DStDevP("[Freight]", "Orders", "[Ship Country] = 'UK'") ' Population.
```

The next example returns the same estimates using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DStDev("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
Y = DStDevP("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry
➥ & "'")
```

DSum Function

Description

Returns the sum of a set of values in a specified set of records (domain).

Syntax

DSum(*expr*, *domain* [, *criteria*])

Remarks

The **DSum** function uses the following arguments.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to add, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DSum** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DSum** evaluates *expr* against the entire domain.

Unlike **DCount**, which returns the number of records, **DSum** totals the values in a field. For example, you could use **DSum** to determine the total cost of freight charges.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table in the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DSum("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DSum("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use **DSum** in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DSum("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If any records in *domain* are being edited when you use **DSum**, the function uses the most recently saved values.

See Also **DCount** Function, **Sum** Function

Example This example totals the values from the Freight field for orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK`.

```
X = DSum("[Freight]", "Orders", "[Ship Country] = 'UK'")
```

The next example calculates the same total using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DSum("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
```

DVar, DVarP Functions

Description Return estimates of the variance for a population and a population sample represented as a set of values in a specified set of records (domain).

Syntax **DVar**(*expr*, *domain* [, *criteria*])

DVarP(*expr*, *domain* [, *criteria*])

Remarks **DVarP** evaluates a population, and **DVar** evaluates a population sample.

The **DVar** and **DVarP** functions use the following arguments.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to evaluate, or an expression that performs calculations using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other domain aggregate or SQL aggregate functions).
domain	String expression identifying the records that constitute the domain. It can be a table name, a query name, or an SQL expression that returns data.
criteria	Optional string expression used to restrict the range of data on which **DVar** is performed. For example, *criteria* could be the WHERE clause in an SQL expression (without the word WHERE). If *criteria* is omitted, **DVar** evaluates *expr* against the entire domain.

If the *criteria* argument contains non-numeric text other than field names, you must enclose the text in single quotation marks. In the following example from the Orders table of the NWIND.MDB database, `Ship Country` is the name of a field, and `UK` is a string literal.

```
X = DVar("[Freight]", "Orders", "[Ship Country] = 'UK'")
Y = DVarP("[Freight]", "Orders", "[Ship Country] = 'UK'")
Z = DVarP("[Freight]", "Orders", "[Ship Via] = 1")
```

If you use these functions in a macro and *criteria* includes a control name that is also the name of a field in *domain*, you must qualify the name of the control by referring to the form on which it appears. The following example shows how you might specify a field and a control that are both named `Region`.

```
= DVar("[Freight]", "Orders", "[Region] = Form.[Region]")
```

If *domain* refers to fewer than two records, or if fewer than two records satisfy *criteria*, **DVar** and **DVarP** return a **Null** (which indicates that a variance can't be calculated).

If any records in *domain* are being edited when you use **DVar** or **DVarP**, the function uses the most recently saved values.

See Also **Var, VarP** Functions

Example

This example returns estimates of the variance for a population and a population sample for orders shipped to the United Kingdom. The domain is the Orders table in the NWIND.MDB database. The criteria argument restricts the data by making the expression apply only to records in which `Ship Country` equals `UK`.

```
X = DVar("[Freight]", "Orders", "[Ship Country] = 'UK'")      ' Sample.
Y = DVarP("[Freight]", "Orders", "[Ship Country] = 'UK'")     ' Population.
```

The next example returns the same estimates using a replaceable parameter, `SearchCountry`.

```
SearchCountry = "UK"
X = DVar("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
Y = DVarP("[Freight]", "Orders", "[Ship Country] = '" & SearchCountry & "'")
```

Dynaset Property

Applies To

Forms.

Description

Returns a **Dynaset** for a form's underlying records.

This property can be used only in expressions and Access Basic code.

Setting

The Dynaset property has no settings.

Usage

Read-only.

Remarks

The Dynaset property refers to the same underlying query or data as the form. If a form is based on a query, for example, referring to the Dynaset property is the equivalent of cloning a **Dynaset** with the same query. If you then apply a filter to the form, the Dynaset property will reflect the filtering.

Use the Dynaset property when you want to apply a method that can't be used with forms, such as **FindFirst**. The Dynaset property provides access to all the methods and properties that you can use with a **Dynaset**.

You can use the RecordCount property to count the number of records in a **Dynaset**. The following example shows how you can combine the RecordCount property with the Dynaset property to count the records in a form.

```
Forms![Orders].Dynaset.MoveLast
MsgBox "My form contains " & Forms![Orders].Dynaset.RecordCount
➥ & " records.", , "Record Count"
```

Notes

- Unlike its form, the **Dynaset** initially has no current record. To make the same record current in the **Dynaset**, assign the value of the form's Bookmark property to a string variable, and then set the **Dynaset's** Bookmark property to that variable. You can also use one of the Find or Move methods to make any record in the **Dynaset** current.
- You can make the form's current record the same as the **Dynaset's** by setting the form's Bookmark property to the **Dynaset's** Bookmark property.
- If you close the form, or if you change the form's set of underlying records, the **Dynaset** will no longer be valid. Subsequently, if you refer to the **Dynaset** or to previously saved bookmarks in the form or the **Dynaset**, an error will occur.

See Also **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods; **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods

Example This example uses the Dynaset property to create a **Snapshot** that lists all of the fields on the `Orders` form. It then uses a loop to print the names of the fields. Notice that the records are those of the **Snapshot**, not of the form.

```
Sub Print_Field_Names ()
  Dim ListSet As Snapshot
  Set ListSet = Forms![Orders].Dynaset.ListFields()    ' Create Snapshot.
  ListSet.MoveFirst                                    ' Move to first record.
  Do Until ListSet.EOF                                 ' For each record.
    Debug.Print ListSet.Name                           ' Print field name.
    ListSet.MoveNext                                   ' Move to next record.
  Loop
End Sub
```

Echo Action

Description Specifies whether Microsoft Access updates the computer screen while a macro runs. You can use this action to hide or show the results of a macro while it runs.

Action argument	Description
Echo On	Select Yes to turn echo on or No to turn it off. The default is Yes.
Status Bar Text	The text to display in the status bar when echo is turned off. For example, "The macro is running."

Remarks

When Microsoft Access runs a macro, the screen updates often show information not essential to the functioning of the macro. When you set Echo On to No, the macro runs without updating the screen. When the macro finishes, Microsoft Access automatically turns echo back on and repaints the window. The No setting for Echo On doesn't affect the macro or its results.

The Echo action doesn't suppress the display of modal dialog boxes, such as error boxes, or pop-up forms, such as property sheets. You can use these to gather or display information when echo is turned off. Use the SetWarnings action to turn off all modal dialog boxes except error message boxes.

You can run the Echo action more than once in a macro. This allows you to change the status bar text while the macro runs.

If you turn echo off, you can use the Hourglass action to change the mouse pointer into an hourglass to provide a visual indication that the macro is running.

See Also

SetWarnings Action

Access Basic

Syntax

DoCmd Echo *echoon* [, *statusbartext*]

Argument	Description
echoon	Use the reserved word **True** (–1) to turn echo on and **False** (0) to turn it off.
statusbartext	A string expression.

Remarks

If you leave the *statusbartext* argument blank, don't use a comma following the *echoon* argument.

Example

This example uses the Echo action to turn echo off and display the specified text in the status bar while the macro is running.

```
DoCmd Echo False, "The macro is running."
```

Edit Method

Description Copies the current record from a specified **Table** or **Dynaset** to the copy buffer for subsequent editing.

Syntax *object*.**Edit**

Remarks The **Edit** method uses the following argument.

Argument	Description
object	Name of an open **Table** or **Dynaset** object that contains the record you want to edit

Use the **Edit** method to allow changes to be made to the current record. After you make the desired changes to the record, use the **Update** method to save your changes.

Caution If you edit a record and then move to another record without using **Update**, your changes are lost without warning.

Whether the current record is optimistically or pessimistically locked depends on the setting of the object's LockEdits property.

The current record remains current after you use **Edit**.

To use **Edit**, there must be a current record. If there is no current record, or if *object* doesn't refer to an open table or an existing **Dynaset**, an error occurs.

See Also **AddNew** Method, **Delete** Method, LockEdits Property, **Update** Method

Example This example, which uses the Employees table in the NWIND.MDB database, locates each record whose Title field satisfies the search criteria and copies it to the copy buffer. It prepares the record for subsequent editing, changes the job title, and saves the change using the **Update** method.

```
Dim Criteria As String, MyDB As Database, MySet As Dynaset
Dim NewTitle As String
Criteria = "Title = 'Sales Representative'"      ' Set search criteria.
NewTitle = "Account Executive"                   ' Set new job title.
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Employees")      ' Create Dynaset.
MySet.FindFirst Criteria                         ' Find first occurrence.
```

```
Do Until MySet.NoMatch                    ' Loop until no matching records.
  MySet.Edit                              ' Enable editing.
  MySet!Title = NewTitle                  ' Change title.
  MySet.Update                            ' Save changes.
  MySet.FindNext Criteria                 ' Find next occurrence.
Loop                                      ' End of loop.
```

Tip Using an update query to change job titles might be more efficient.

Enabled, Locked Properties

Apply To

Controls (bound object frame, check box, combo box, command button, list box, option button, option group, subform, text box, toggle button, unbound object frame) on a form. The Locked property doesn't apply to the command button or unbound object frame.

Description

- Enabled—determines whether a control can have the focus in Form view.
- Locked—determines whether a control allows changes to data in Form view.

Setting

The Enabled property settings are:

Setting	Description
Yes	(Default for all controls except unbound object frame) The control can have the focus. If the control is an object frame, double-clicking it executes the control's primary verb; for example, it will play an embedded sound object.
No	(Default for unbound object frame) The control can't have the focus and appears dimmed. If the control is an option group, neither the option group nor the controls inside the option group can have the focus.

The Locked property settings are:

Setting	Description
Yes	The control functions normally but doesn't allow additions or changes to data.
No	(Default) The control functions normally and allows additions or changes to data.

Remarks

Use the Enabled property to enable and disable controls. For example, in Form view you can disable a command button until you have changed data in a text box. You can then use the textbox's AfterUpdate property to call a macro to enable the command button.

Use the Locked property to protect data in a field by making it read-only. For example, you might want a field to only display information or you might want to lock a field until a specific condition is met.

The following table shows all combinations of the Enabled and Locked property settings.

Enabled	Locked	Description
Yes	Yes	Control can have the focus. Data is displayed normally and can be copied but not changed.
No	No	Control can't have the focus. Control and data appear dimmed.
Yes	No	Control can have the focus. Data is displayed normally and can be copied and changed.
No	Yes	Control can't have the focus. Data is displayed normally but can't be copied or changed.

Access Basic

The Enabled and Locked settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read/write.

See Also DisplayWhen Property, SetValue Action, Visible Property

End Statement

Description Ends an Access Basic procedure or block.

Syntax **End** [{**Function** | **If** | **Select** | **Sub** | **Type**}]

Remarks You can use the **End** statement in the following ways.

Statement	Description
End Function	Ends a **Function** procedure definition. You must use **End Function** with **Function**. Automatically created when you start a new **Function** procedure.
End If	Ends a block **If...Then** statement. Required to end a block **If...Then...Else.**
End Select	Required to end a **Select Case** block.

Statement	Description
End Sub	Required to end a **Sub** procedure. Automatically created when you start a new **Sub** procedure.
End Type	Required to end a user-defined type definition (**Type** statement).
End	By itself, terminates program execution. Equivalent to choosing Reinitialize from the Module window Run menu. Never required and may be placed anywhere.

See Also **Exit** Statement, **Function** Statement, **If...Then...Else** Statement, **Select Case** Statement, **Stop** Statement, **Sub** Statement, **Type** Statement

Example This example uses **End** to end code execution, to end a **Select Case** block, and to end a **Sub** procedure.

```
Sub Get_Name ()
  Do
    MyName = InputBox("Enter your name, or enter X to end the program.")
    Select Case MyName                    ' Evaluate input.
      Case "X", "x"
        Exit Do                           ' Exit loop.
      Case "A" To "Z", "a" To "z"
        MsgBox "Your name is " & MyName & "."
      Case Else
        MsgBox "You didn't enter your name or X."
    End Select                            ' End Select Case block.
  Loop
  End
End Sub                                   ' End Sub procedure.
```

Environ, Environ$ Functions

Description Return the string associated with an operating system environment variable.

Syntax 1 **Environ**[**$**](*environmentstring*)

Syntax 2 **Environ**[**$**](*n*)

Remarks **Environ** returns a **Variant**; **Environ$** returns a **String**.

The **Environ**[$] function uses these arguments.

Argument	Description
environmentstring	A string expression containing the name of an environment variable.
n	A numeric expression corresponding to the *n*th string of the environment-string table. The argument *n* can be any numeric expression, but it is rounded to an integer before it is evaluated.

If the environment variable name you specify cannot be found in the environment-string table, a zero-length **String** or **Variant** of **VarType** 8 is returned. Otherwise, **Environ**[$] returns the text assigned to the specified environment variable (the text following the equal sign in the environment-string table for that environment variable.)

If you specify a numeric argument (*n*), the *n*th string of the environment-string table is returned. In this case, **Environ**[$] returns all of the text, including *environmentstring*. If the *n*th string doesn't exist, **Environ**[$] returns a zero-length **String** or **Variant** of **VarType** 8.

Example

This example uses **Environ** to supply the entry number and length of the PATH statement from the environment-string table.

```
Indx = 1                                        ' Initialize Index to 1.
Do
  EnvString = Environ(Indx)                     ' Get environment variable.
  If Left(EnvString, 5) = "PATH=" Then          ' Check PATH entry.
    PathLen = Len(Environ("PATH"))              ' Get length.
    Msg = "PATH entry = " & Indx & " and length = " & PathLen
    Exit Do
  Else
    Indx = Indx + 1                             ' Not PATH entry, so increment.
  End If
Loop Until EnvString = ""
If PathLen > 0 Then
  MsgBox Msg                                    ' Display message.
Else
  MsgBox "No PATH statement exists."
End If
```

EOF Function

Description Returns a value during file input that indicates whether or not the end of a file has been reached.

Syntax **EOF**(*filenumber*)

Remarks The argument *filenumber* is the number used in the **Open** statement to open the file. You can use any numeric expression that evaluates to the number of an open file.

The **EOF** function returns **True** if the end of a file has been reached. Use the **EOF** function with sequential files to test for the end of a file. This helps you avoid the error that occurs when you attempt to get input past the end of a file.

When used with **Random** or **Binary** files, **EOF** returns **True** if the last executed **Get** statement was unable to read an entire record; otherwise, it returns **False**.

See Also **Loc** Function, **LOF** Function, **Open** Statement

Example This example uses the **EOF** function to detect the end of a sequential file.

```
Open "TESTFILE" For Input As #1          ' Open file for input.
Do While Not EOF(1)                      ' Check for end of file.
  Input #1, TestVariable                 ' Read data.
Loop
Close #1                                 ' Close data file.
```

EOF Property

Applies To Recordsets.

Description Indicates whether the current record position is after the last record in a recordset (at the end of a file).

Setting The EOF property settings are:

Setting	Description
–1	The current record position is after the last record in a recordset.
0	The current record position is on or before the last record in a recordset.

Usage Read-only.

Remarks You can use the EOF property to determine whether you have gone beyond the limits of a recordset as you move from record to record.

If the last record is the current record when you use the **MoveNext** method, EOF is set to **True** (–1). If you use **MoveNext** while EOF is **True**, an error occurs; EOF remains **True** and there is no current record.

When you create or open a recordset, the first record is the current record and EOF is **False** (0). It remains **False** until you move past the last record in the recordset. If the recordset contains no records, EOF is **True**.

See Also BOF Property, NoMatch Property

Example See BOF Property

Eqv Operator

Description Used to perform a logical equivalence on two expressions.

Syntax *result* = *expr1* **Eqv** *expr2*

Remarks If either expression is a **Null**, *result* is also a **Null**. When neither expression is a **Null**, *result* is determined according to the following table.

If *expr1* is	And *expr2* is	*result* is
true (nonzero)	true	**True** (–1)
true	false (0)	**False** (0)
false	true	**False**
false	false	**True**

The **Eqv** operator performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

If bit in *expr1* is	And bit in *expr2* is	*result* is
0	0	1
0	1	0
1	0	0
1	1	1

Bit-wise comparisons can be performed only in Access Basic code.

See Also **And** Operator, Comparison Operators, **Imp** Operator, **Not** Operator, Operator Precedence, **Or** Operator, **Xor** Operator

Example This example prints a message that depends on the value of variables `A`, `B`, and `C`, assuming that no variable is a **Null**. If `A` = 10, `B` = 8, and `C` = 6, both expressions evaluate **True**. As a result, the **Eqv** expression also evaluates **True**.

```
If A > B Eqv B > C Then
  Debug.Print "Both expressions are True or both are False."
Else
  Debug.Print "One expression is True and one is False."
End If
```

Erase Statement

Description Reinitializes the elements of fixed arrays and deallocates dynamic-array storage space.

Syntax **Erase** *arrayname* [, *arrayname*] . . .

Remarks The argument *arrayname* is the name of the array to be erased. It is important to know whether an array is fixed (ordinary) or dynamic because **Erase** behaves differently depending on the type of array. For fixed arrays, no memory is recovered. **Erase** sets the elements of a fixed array as follows.

Type of array	Effect of Erase on array elements
Fixed numeric array	Sets each element to zero
Fixed string array (variable-length)	Sets each element to zero-length ("")
Fixed string array (fixed-length)	Sets each element to zero
Fixed **Variant** array	Sets each element to Empty
Array of user-defined types	Sets each element as if it were a separate variable

For dynamic arrays, **Erase** frees the memory used by the array. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a **ReDim** statement.

See Also **Dim** Statement, **ReDim** Statement

Example In this example, a two-dimensional array is created, filled, and erased. When the **Erase** statement is executed, zeros replace the contents of each element, but no memory space is recovered.

```
Static Matrix(50, 50) As Integer          ' Create 2-D array.
For I = 0 To 50
  For J = 0 To 50
    Matrix(I, J) = J                      ' Put some values into array.
  Next J
Next I
Erase Matrix                              ' Erase array.
```

Erl Function

See **Err**, **Erl** Functions

Err, Erl Functions

Description Return error status.

Syntax **Err**

Erl

Remarks After an error occurs, the **Err** function returns an **Integer** run-time error code identifying the error. The **Erl** function returns an **Integer** that is the line number of the line in which the error occurred or the line most closely preceding it.

Because **Err** and **Erl** return meaningful values only after an error has occurred, they are usually used in error-handling routines to determine the error and the corrective action. Both **Err** and **Erl** are reset to 0 after any form of the **Resume** or **On Error** statement and after an **Exit Sub** or **Exit Function** statement within an error-handling routine.

Caution If you set up an error handler using **On Error GoTo** and that error handler calls another procedure, the value of **Err** and **Erl** may be reset to 0. To make sure that the value doesn't change, assign the values of **Err** or **Erl** to variables before calling another procedure or before executing **Resume**, **On Error**, **Exit Sub**, or **Exit Function.**

You can directly set the value returned by the **Err** function using the **Err** statement. You can set the values for both **Err** and **Erl** indirectly using the **Error** statement.

The **Erl** function returns only a line number, not a line label, located at or before the line producing the error. Line numbers greater than 65,529 are treated as line labels and can't be returned by **Erl**. If your program has no line numbers, or if there is no line number before the point at which an error occurs, **Erl** returns 0.

See Also **Err** Statement, **Error** Statement, **On Error** Statement, **Resume** Statement

Example This example shows a typical error-handling routine. If there are no additional errors, **Err** returns error number 11 (`Division by zero`) and **Erl** returns line `1020`.

```
Sub ErrDemo
Dim A As Integer, B As Integer, C As Integer
1000  On Error GoTo ErrHandler           ' Set up an error handler.
1010  B = 1
1020  A = B \ C                          ' Cause a Division by zero error.
1030  Exit Sub

ErrHandler:
  MsgBox "Error number " & Err & " occurred at line " & Erl
  Resume Next
End Sub
```

Err Statement

Description Sets **Err** to a specific value.

Syntax **Err** = *n*

Remarks The argument *n* is either an integer expression between 1 and 32,767, inclusive, that specifies a run-time error code or 0, which means no run-time error has occurred.

When running an application, Access Basic uses **Err** to record whether or not a run-time error has occurred and what the error was.

Use the **Err** statement to set **Err** to a nonzero value to communicate error information between procedures. For example, you might use one of the run-time error codes not used by Access Basic as an application-specific error code. To determine which error codes are being used, use the **Error**[$] function and/or the **Error** statement. You can create application-specific user-defined errors by working down from error code 32,767.

You can also set **Err** to 0 by using any form of the **Resume** or **On Error** statement or by executing an **Exit Sub** or **Exit Function** statement within an error handler. In addition, the **Error** statement can set **Err** to any value to simulate any run-time error.

See Also **Err, Erl** Functions; **Error, Error$** Functions; **Error** Statement

Example

As this example shows, with **On Error Resume Next**, errors are often handled in the lines of code immediately following the line in which an error can occur. Once the error has been handled, the **Err** statement clears the error by setting **Err** to 0 so that subsequent error-handling code isn't inadvertently executed for an earlier error.

```
On Error Resume Next                                    ' Set up error handler.
FileName = InputBox("Enter a file name: ")              ' Get file name from user.
Do
  Open FileName For Input As #1                         ' Open file.
  Select Case Err                                       ' Handle error, if any.
    Case 0                                              ' No error.
    Case 53, 75, 76                                     ' Assorted file errors.
      FileName = InputBox("File not found. Enter another file name: ")
    Case 71                                             ' Drive not ready.
      MsgBox "Close the drive door."
    Case Else
      MsgBox "Cannot continue!"                         ' All fatal errors.
      Stop
  End Select
  Err = 0                                               ' Clear error.
Loop
```

Error, Error$ Functions

Description

Return the error message that corresponds to a given error code.

Syntax

Error[**$**][(*errorcode*)]

Remarks

Error returns a **Variant**; **Error$** returns a **String**.

The argument *errorcode* must be an integer between 1 and 32,767, inclusive. If *errorcode* is omitted, the message string corresponding to the most recent run-time error is returned. If *errorcode* is not defined in Access Basic, the returned message string is `User-defined error`.

Some error messages use internal variables to provide specific information about an error. When an error occurs, words appropriate to the context of the error are substituted in the error message string. Under normal circumstances, the following error message appears with a file name inserted between the single quotation marks:

```
Not a valid document name: 'TESTFILE.DOC'.
```

However, when you use **Error**[**$**] to return an error message for a specified error code, there is no context for that error message and the appropriate text can't be inserted. As a

result, the message string returned by **Error**[$] contains no context-specific information between the single quotation marks. A space is inserted in lieu of the actual context information.

If **Error**[$] is called with no argument, it returns the message string for the most recent run-time error. If a run-time error has occurred, the returned message string includes substitutions appropriate to the context of the error. If no run-time error has occurred, **Error**[$] returns a zero-length string ("").

The **Err** function returns the error code for the most recent run-time error. Therefore, the following lines are equivalent except that in the first case, the text placed in `A` includes appropriate substitutions, and in the second case, no substitutions are made.

```
A = Error
A = Error(Err)
```

See Also **Err, Erl** Functions; **Error** Statement

Example This example uses the **Error** function to display error messages associated with user-input error numbers.

```
ErrNum = InputBox("Enter an error number to see its error message.")
MsgBox = "Error number is: " & ErrNum & "; message is: " & Error(ErrNum)
```

Error Statement

Description Simulates the occurrence of an error; can be used to generate a user-defined error.

Syntax **Error** *errorcode*

Remarks The argument *errorcode* must be an integer between 1 and 32,767, inclusive. If *errorcode* is an error code defined by Access Basic, the **Error** statement simulates the occurrence of that error; that is, it sets the value of **Err** to *errorcode*.

To define your own error code, use a value that is greater than any used by the standard Access Basic error codes. You can create application-specific user-defined errors working down from error code 32,767.

If an **Error** statement is executed when no error-handling routine is enabled, Access Basic displays an error message and stops program execution. If the **Error** statement specifies an error code that is not used by Access Basic, the message `User-defined error` is displayed.

See Also **Err, Erl** Functions; **On Error** Statement; **Resume** Statement

Example In this example, the **Error** statement simulates the occurrence of an error so that the error handler is invoked and the associated error message is displayed.

```
Sub Error_Test

  On Error GoTo ErrorHandler          ' Set up error handler.
  NL = Chr(13) & Chr(10)              ' Define newline.
  Msg = "Enter an error number to see the associated error message."
  ErrNum = InputBox(Msg)
  Error ErrNum                        ' Simulate error occurrence.
  Exit Sub

ErrorHandler:       ' Beginning of error handler.
  Msg = "The error message for error number "
  Msg = Msg & Err & " is: " & NL & NL
  Msg = Msg & """" & Error(Err) & """"
  MsgBox Msg        ' Display message.
  Resume Next

End Sub
```

Eval Function

Description Evaluates an expression and returns its value.

Syntax **Eval**(*stringexpr*)

Remarks The **Eval** function uses the following argument.

Argument	Description
stringexpr	Expression that results in a text string.

You can use the **Eval** function in fields on a form or report or in Access Basic to evaluate text and return the evaluation. The *stringexpr* must evaluate to a string or numeric value; **Eval** cannot return a Microsoft Access object.

You can construct a string and then pass it to **Eval** for evaluation as if the string were an actual expression. For example, `Eval("1 + 1")` returns `2`. The next example, which is perhaps more useful, prompts the user for a control name, returns the value of that control, and then tells whether the control is visible.

```
X = InputBox("Enter the control name.")
MsgBox ("The current value of " & X & " is " & Eval(X))
MsgBox (X & " is " & IIf(Eval(X & ".Visible"), "", "in") & "visible.")
```

You can use the **Eval** function to access expression features that aren't ordinarily available in Access Basic. For example, you cannot use the **Between...And**, **In**, and **Like** operators directly, but you can use them with **Eval**. The next example determines whether the value in the Ship Region field is one of several specified Australian state abbreviations. It also shows how you can use single quotation marks to include one string within another string. If the field contains one of the abbreviations, `IsAusState` will be **True** (–1).

```
IsAusState = Eval("Forms!Orders![Ship Region] In ('NSW', 'QLD', 'SA',
➥  'WA', 'NT', 'VIC', 'TAS')")
```

See Also **Choose** Function, **IIf** Function, **Switch** Function

Example The following example creates a simple calculator. It accepts input from the user with the **InputBox** function. **Eval** evaluates the input and returns a value that **MsgBox** displays.

```
Sub Calculator
  Dim CRLF As String, Result As Variant, T As String
  CRLF = Chr$(13) + Chr$(10)
  T = Chr$(9)
  Result = InputBox("Enter a numeric expression.")
  MsgBox "Your expression:" + T + Result + CRLF + "Evaluates to:" + T +
  ➥  Str$(Eval(Result)), 0, "Calculation Result"
End Sub
```

The next example triggers an OnPush event as if the user had clicked a button on a form. If the value of the button's OnPush property begins with an equal sign (that is, it is the name of a function), **Eval** triggers the event. Otherwise, the value names a macro, which the **DoCmd** statement runs.

```
Dim MyControl As Control, Temp As Variant
Set MyControl = Forms!MyForm!MyButton
If (Left(MyControl.OnPush, 1) = "=") Then
  Temp = Eval(Mid(MyControl.OnPush,2))
Else
  DoCmd RunMacro MyControl.OnPush
EndIf
```

Execute Method

Description Invokes an action **QueryDef** in a specified database.

Syntax *querydef*.**Execute**

Remarks The **Execute** method uses the following argument.

Argument	Description
querydef	Name of the object that refers to the **QueryDef** you want to invoke

The **Execute** method is valid only for action queries. If you use **Execute** on a nonaction query, an error occurs.

Because an action query doesn't return any rows, **Execute** doesn't return a recordset.

If *querydef* doesn't refer to an open **QueryDef**, an error occurs.

See Also **CreateDynaset** Method, **CreateQueryDef** Method, **CreateSnapshot** Method, **OpenQueryDef** Method

Example This example creates a query that changes the title of all sales representatives in the Employees table of the NWIND.MDB database.

```
Dim MyDB As Database, MyQuery As QueryDef
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("Change Job Titles")    ' Create query.
' Set SQL property.
MyQuery.SQL = "UPDATE DISTINCTROW Employees SET Employees!Title =
➥ 'Account Executive' WHERE Employees!Title = 'Sales Representative';"
MyQuery.Execute                                           ' Invoke query.
MyQuery.Close                                             ' Close query.
MyDB.DeleteQueryDef("Change Job Titles")                  ' Delete query.
```

Exit Statement

Description Exits a **Do...Loop**, a **For...Next** loop, a **Function** procedure, or a **Sub** procedure.

Syntax **Exit { Do | For | Function | Sub }**

Remarks The **Exit** statement is used as follows.

Statement	Description
Exit Do	Provides a way to exit a **Do...Loop** statement. It can be used only inside a **Do...Loop** statement. **Exit Do** transfers control to the statement following the **Loop** statement. When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop in which it occurs.
Exit For	Provides a way to exit a **For...Next** loop. It can be used only in a **For...Next** loop. **Exit For** transfers control to the statement following the **Next** statement. When used within nested **For...Next** loops, **Exit For** transfers control to the loop that is one nested level above the loop in which it occurs.
Exit Function, Exit Sub	Immediately exit the procedure in which they appear. **Exit** statement must match procedure type. Program execution continues with the statement following the statement that has called the procedure.

Don't confuse **Exit** statements with **End** statements. **Exit** doesn't define the end of a structure.

See Also **Do...Loop** Statement, **End** Statement, **For...Next** Statement, **Function** Statement, **Stop** Statement, **Sub** Statement

Example This example uses **Exit** to exit a **For...Next** loop, a **Do...Loop**, and a **Sub** procedure.

```
Sub Exit_Demo ()
  Do                                    ' Set up infinite loop.
    For Indx = 1 To 1000
    Num = Int(Rnd * 100)                ' Generate random numbers.
      Select Case Num                   ' Evaluate random number.
        Case 7 : Exit For               ' If 7, exit For...Next.
        Case 29 : Exit Do               ' If 29, exit Do...Loop.
        Case 54 : Exit Sub              ' If 54, exit Sub procedure.
      End Select
    Next Indx
  Loop
End Sub
```

Exp Function

Description Returns *e* (the base of natural logarithms) raised to a power.

Syntax **Exp**(*number*)

Remarks If the value of *number* exceeds 709.782712893, an `Overflow` error occurs. The constant *e* approximately equals 2.718282.

Note The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.

See Also **Log** Function

Example This example uses **Exp** to calculate the value of *e*. **Exp**(1) is *e* raised to the power of 1.

```
= Exp(1)
```

FieldName Property

Applies To Table fields.

Description Specifies the name of a field.

Setting Enter a valid name. The name can't duplicate any other FieldName in the table.

Remarks Microsoft Access identifies a field by its FieldName. You can use the FieldName in expressions, macros, procedures, and SQL statements.

Access Basic This property is not available from Access Basic.

To obtain information about table fields, including field names, use the **ListFields** method.

See Also Caption Property

FieldSize Method

Description Returns the number of bytes in a Memo or an OLE Object field.

Syntax *varname* = *object*.**FieldSize( )**

Remarks The **FieldSize** method uses the following arguments.

Argument	Description
varname	Name of a **Long** or **Variant** variable
object	Name of a Memo or an OLE Object field

Because the size of an OLE Object field can exceed 64K, you should assign the value returned by the **FieldSize** method to a variable capable of storing a **Long** variable. Because the size of a Memo field is limited to 32K bytes, you can assign the value returned by the **FieldSize** method to an **Integer** variable.

You can use **FieldSize** with the **AppendChunk** and **GetChunk** methods to manipulate large fields.

See Also **AppendChunk** Method, **GetChunk** Method, **ListFields** Method

Example This example uses the **GetChunk** method to return consecutive 16K chunks of the Notes field in the Employees table of the NWIND.MDB database. It uses **FieldSize** to determine how many chunks it needs to copy and copies the chunks to the string array `NoteArray( )`, where they can be changed. **AppendChunk** then copies `NoteArray( )` back to `Notes`.

```
Option Base (1) ' This line should appear in the Declarations section.

Sub Get_Notes
  Const ChunkSize = 16384                          ' Set size of chunk.
  Dim NumChunks As Integer, TotalSize As Long
  Dim MyDB As Database, MyTable As Table, X As Integer
  Set MyDB = CurrentDB()
  Set MyTable = MyDB.OpenTable("EMPLOYEES")        ' Open table.
  TotalSize = MyTable!Notes.FieldSize ()           ' Get field size.
   ' How many chunks?
  NumChunks = TotalSize \ ChunkSize - (TotalSize Mod ChunkSize <> 0)
  ReDim NoteArray(NumChunks) As String * ChunkSize
  ' Get current Notes field.
  For X = 1 To NumChunks
    NoteArray(X) = MyTable!Notes.GetChunk((X - 1) * ChunkSize, ChunkSize)
  Next X
  ...                                              ' Make changes.
```

```
MyTable.Edit                                          ' Enable editing.
  MyTable!Notes = ""                                  ' Initialize Notes field.
  For X = 1 To NumChunks
    MyTable!Notes.AppendChunk(NoteArray(X))           ' Replace with edited Notes.
  Next X
  MyTable.Update                                      ' Save changes.
  MyTable.Close                                       ' Close table.
End Sub
```

FieldSize Property

Applies To Table fields.

Description Sets the maximum size of data that can be stored in a field.

Setting If the DataType property is set to Text, enter a number less than 255. The default setting is 50.

If the DataType property is set to Number, the FieldSize property settings and their values are related in the following way.

Setting	Description
Byte	Stores numbers from 0 to 255 (no fractions). It occupies 1 byte.
Integer	Stores numbers from –32,768 to 32,767 (no fractions). It occupies 2 bytes.
Long Integer	Stores numbers from –2,147,483,648 to 2,147,483,647 (no fractions). It occupies 4 bytes.
Single	Stores numbers with six digits of precision, from –3.402823E38 to 3.402823E38. It occupies 4 bytes.
Double	(Default) Stores numbers with ten digits of precision, from –1.79769313486232E308 to 1.79769313486232E308. It occupies 8 bytes.

Remarks You should generally use the smallest possible FieldSize setting because smaller data sizes can be processed faster and require less memory.

Caution If you convert a large FieldSize setting to a smaller one in a table that already contains data, you might lose data. For example, if you change the FieldSize setting for a Text field from 255 to 50, data over 50 characters long is truncated.

If the data in a Number field doesn't fit in a new setting range, fractional numbers may be rounded or you might get an empty field. For example, if you change from Single to Integer, fractional values are rounded to the nearest whole number and values greater than 32,768 or less than –32,767 result in empty fields.

You can't undo changes to a table's design after saving it in Design view.

Note Use the Currency data type if you plan to perform many calculations on a field that contains data with one to four decimal places. Single and Double fields require floating-point calculation. Currency fields use a faster fixed-point calculation that avoids rounding errors.

Access Basic

The FieldSize property is available using the **ListFields** method.

For more information about Access Basic data types, see Access Basic Data Types.

FileAttr Function

Description

Returns file mode or operating system file handle information about an open file.

Syntax

FileAttr(*filenumber*, *attribute*)

Remarks

The **FileAttr** function uses these arguments.

Argument	Description
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.
attribute	Number that indicates the type of information to return about the file. When *attribute* is 2, **FileAttr** returns the operating system handle for the file. When *attribute* is 1, **FileAttr** returns one of the following values to indicate the file's mode:

Return value	Mode
1	Input
2	Output
4	Random
8	Append
32	Binary

See Also **Open** Statement

Example This example uses **FileAttr** to report the file handle and mode of an open file.

```
FileNum = FreeFile                              ' Get available file number.
Open "TESTFILE" For Append As FileNum
Handle = FileAttr(FileNum,2)                    ' Get file handle.
Select Case FileAttr(FileNum,1)                 ' Determine mode.
  Case 1 : Mode = "Input"
  Case 2 : Mode = "Output"
  Case 4 : Mode = "Random"
  Case 8 : Mode = "Append"
  Case 32 : Mode = "Binary"
End Select
Close FileNum                                   ' Close file.
MsgBox "File " & Handle & " was opened for " & Mode
```

FillColor Property

Applies To Reports.

Description Specifies the color that fills in boxes and circles drawn with the **Line** and **Circle** methods.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting Use a numeric expression with the **RGB** or **QBColor** function to set the value of this property.

Usage The value of this property is a **Long** data type.

Design view: not available. Other views: read/write.

Remarks Use this property with Access Basic to create special visual effects on custom reports when you print with a color printer.

See Also DrawMode Property, DrawStyle Property, DrawWidth Property, OnFormat Property, OnPrint Property, **PSet** Method

FillStyle Property

Applies To Reports.

Description Determines whether a circle or line drawn by the **Circle** or **Line** method is transparent.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting The FillStyle property settings are:

Setting	Description
1	(Default) Normal—the circle or line has the color set by the FillColor property.
0	Clear—the circle or line is transparent and has the color of the form or report behind it.

Usage Design view: read/write. Other views: read-only.

Remarks To use the FillStyle property, the SpecialEffect property must be set to Normal.

See Also BorderColor Property, BorderStyle Property, BorderWidth Property

Filter Property

Applies To **Dynasets**, **Snapshots**.

Description Sets a filter when you create a **Dynaset** or **Snapshot.**

Setting The Filter property setting is the WHERE expression in an SQL string (without the word WHERE) applied to the records.

Usage Read/write.

Remarks You can use the Filter property to select records from a table. The following example shows how you can select only the records for customers who live in a particular region:

```
MyDynaset.Filter = "[Region] = 'NY'"
```

You can't use the Filter property with a **Snapshot** created with one of the List methods.

See Also **CreateDynaset** Method, **CreateSnapshot** Method

Example This example creates a **Dynaset** (based on the Orders table in the NWIND.MDB database) that contains only records for orders shipped to the United Kingdom.

```
Dim MyDB As Database, MySet As Dynaset, FilteredSet As Dynaset
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Orders")  ' Create Dynaset.
MySet.Filter = "[Ship Country] = 'UK'"    ' Set filter condition.
Set FilteredSet = MySet.CreateDynaset() ' Create filtered Dynaset.
```

Note To change the contents of `MySet`, you cannot reset the Filter property. Instead, you must create a second **Dynaset** or **Snapshot**.

Tip It might be more efficient to create the second **Dynaset** with the desired conditions in one step. As a general rule, when you know which data you want to select, it is usually better to create a **Dynaset** with an SQL string. The next example shows how you can create just one **Dynaset** and obtain the same results as in the preceding example.

```
Dim MyDB As Database, MySet As Dynaset
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("SELECT * FROM Orders WHERE [Ship Country]
➥ = 'UK';")
```

FindFirst, FindLast, FindNext, FindPrevious Methods

Description Locate the first, last, next, or previous record that satisfies the specified criteria and makes that record the current record.

Syntax

object.**FindFirst** *criteria*

object.**FindLast** *criteria*

object.**FindNext** *criteria*

object.**FindPrevious** *criteria*

Remarks The Find methods use the following arguments.

Argument	Description
object	Name of an existing **Dynaset** or **Snapshot**
criteria	String expression (the WHERE clause in an SQL string without the word WHERE) used as the Find condition

Use the Find methods to locate records in a **Dynaset** or **Snapshot** that satisfy a specific condition. If you want to include all the records in your search—not just those that meet a specific condition—use the Move methods to move from record to record. To locate a record in a **Table**, use the **Seek** method.

If *object* contains more than one record that satisfies *criteria*, **FindFirst** locates the first occurrence. **FindNext** locates the next one, and so on. Using one of these Find methods isn't the same as using **MoveFirst** or **MoveNext**, however, which simply makes the first or next record current without applying a condition. You can follow a Find operation with a Move operation.

You can't use the Find methods with a **Snapshot** created with the **ListFields**, **ListIndexes**, **ListParameters**, or **ListTables** method.

Note Always inspect the value of the NoMatch property to determine whether the Find operation has succeeded. If it fails, NoMatch will be **True** and the current record will be undefined.

See Also **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods; NoMatch Property; **Seek** Method

Example This example changes the job title of all sales representatives in the Employees table of the NWIND.MDB database. It creates a **Dynaset** and then uses **FindFirst** and **FindNext** to locate all records satisfying the title condition. It prepares each record for editing, changes the title, and saves the change with **Update**.

```
Dim Criteria As String, MyDB As Database, MySet As Dynaset
Criteria = "Title = 'Sales Representative'"  ' Define search criteria.
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Employees")  ' Create Dynaset.
MySet.FindFirst Criteria                     ' Locate first occurrence.
Do Until MySet.NoMatch                      ' Loop until no matching records.
  MySet.Edit                                ' Enable editing.
  MySet!Title = "Account Executive"         ' Change title.
  MySet.Update                              ' Save changes.
  MySet.FindNext Criteria                   ' Locate next record.
Loop                                        ' End of loop.
```

Tips

- You could locate the matching records in reverse order by finding the last occurrence with **FindLast** and then using **FindPrevious** instead of **FindNext**.
- Using an update query to change job titles might be more efficient.

FindLast Method

See **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods

FindNext Action

Description Finds the next record that meets the criteria specified by the previous FindRecord action or in the Find dialog box of the Find command.

Remarks You can use the FindNext action to search repeatedly for records. For example, you can move successively through all the records for a particular customer.

To set the search criteria, use the FindRecord action. Typically, you enter a FindRecord action in a macro and then use the FindNext action to find succeeding records that meet the same criteria. To search for records only when a certain condition is met, you can enter a conditional expression in the Condition column of the FindNext action.

This action has the same effect as using the Find Next button in the Find dialog box.

The arguments for the FindRecord action are shared with the options in the Find dialog box. The FindNext action finds the next record that meets the Find criteria as set either by the FindRecord action or in the Find dialog box.

Tip If you've set the Search In argument of the FindRecord action to Current Field, you may need to use the GoToControl action to move the focus to the field containing the data you're searching for before you use the FindNext action.

See Also FindRecord Action

Access Basic

Syntax

DoCmd FindNext

Remarks

This action takes no arguments.

FindNext Method

See **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods

FindPrevious Method

See **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods

FindRecord Action

Description Finds the first record following the current record that meets the criteria specified by the arguments. You can find records in the active table datasheet, form datasheet, form, or query dynaset.

Action argument	Description
Find What	The data you want to find in the record. Enter the text, number, or date you want to find or type an expression (preceded by an equal sign). You can use wildcard characters. Required argument.
Where	Specifies where the data is located in the field. You can specify a search for data in any part of the field (Any Part of Field), for data that fills the entire field (Match Whole Field), or for data located at the beginning of the field (Start of Field). The default is Match Whole Field.
Match Case	Specifies whether the search is case-sensitive (uppercase and lowercase letters must match exactly). Select Yes or No. The default is No.
Direction	Specifies whether the search proceeds from the current record up to the beginning or down to the end of the records. Select Up or Down. The default is Down.
Search As Formatted	Specifies whether the search includes formatted data. If you select Yes, Microsoft Access searches for the data as it is formatted and displayed in the field. If you select No, Microsoft Access searches for the data as it is stored in the database, which is not always the same as it is displayed. The default is No. You can use this feature to restrict the search to data in a particular format. For example, select Yes and enter 1,234 in the Find What argument to find a value of 1,234 in a field formatted to include commas. Select No if you want to enter 1234 to search for the data in this field.

Action argument	Description
	To search for dates, select Yes to find a date exactly as it is formatted, such as 03-March-1991. If you select No, enter the date in the Find What argument in the format set in the International section of the Microsoft Windows Control Panel. (This is the Short Date format set in the Date Format section.) For example, if the Short Date format is set to m/d/yy, you can enter 3/9/91, and Microsoft Access will find all entries in a Date field that correspond to March 9, 1991, regardless of how this field is formatted. **Note** This argument takes effect only if the current field is a bound field, the Where argument is set to Match Whole Field, the Search In argument is set to Current Field, and the Match Case argument is set to No.
Search In	Specifies whether the search is confined to the current field in each record or includes all fields in each record. The Current Field search is faster. Select Current Field or All Fields. The default is Current Field.
Find First	Specifies whether the search starts at the first record or at the current record. Select Yes to start at the first record. Select No to start at the current record. The default is Yes.

Remarks

When a macro runs the FindRecord action, Microsoft Access moves to the next record containing the specified data and selects the data in the record.

The FindRecord action is equivalent to choosing the Find command from the Edit menu, and its arguments are the same as the options in the Find dialog box. If you set the FindRecord action arguments in the Macro window and then run the macro, you'll see the corresponding options selected in the Find dialog box when you choose the Find command.

Microsoft Access retains the most recent FindRecord arguments during a database session so that you don't need to enter the same criteria repeatedly as you perform subsequent FindRecord operations. If you leave an argument blank, Microsoft Access uses the most recent setting for the argument, as set either by a previous FindRecord action or in the Find dialog box.

See Also

FindNext Action

Access Basic

Syntax

DoCmd FindRecord *findwhat* [, *where*] [, *matchcase*] [, *direction*] [, *searchasformatted*] ➥ [, *searchin*] [, *findfirst*]

Argument	Description
findwhat	An expression that evaluates to text, a number, or a date. The expression contains the data to search for.
where	One of the following intrinsic constants: A_ANYWHERE A_ENTIRE A_START If you leave this argument blank, the default (A_ENTIRE) is assumed.
matchcase	Use the reserved word **True** (–1) for a case-sensitive search and **False** (0) for a search that is not case-sensitive. If you leave this argument blank, the default (**False**) is assumed.
direction	One of the following intrinsic constants: A_UP A_DOWN If you leave this argument blank, the default (A_DOWN) is assumed.
searchasformatted	Use the reserved word **True** to search for data as it is formatted and the reserved word **False** to search for data as it is stored in the database. If you leave this argument blank, the default (**False**) is assumed.
searchin	One of the following intrinsic constants: A_CURRENT A_ALL If you leave this argument blank, the default (A_CURRENT) is assumed.
findfirst	Use the reserved word **True** to start the search at the first record. Use the reserved word **False** to start the search at the record following the current record. If you leave this argument blank, the default (**True**) is assumed.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example This example finds the first occurrence in the records of the name `Smith` in the current field. It doesn't find occurrences of smith or Smithson.

```
DoCmd FindRecord "Smith", , True
```

First, Last Functions

Description Return a field value from the first or last record in a query, form, or report.

Syntax **First**(*expr*)

Last(*expr*)

Remarks The **First** and **Last** functions use the following argument.

Argument	Description
expr	String expression identifying the field that contains the data you want to use, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

You can use **First** and **Last** to determine the limits of the underlying query. For example, the records returned by **First** and **Last** might contain order dates that you could use in a report title to identify the range for orders.

You can use **First** and **Last** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **Count** Function; **DFirst**, **DLast** Functions; **Min**, **Max** Functions; **StDev**, **StDevP** Functions; **Sum** Function; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to return the values from the Shipped Date field of the first and last records for orders shipped to the United Kingdom. You can enter these expressions in the SQL dialog box in the Query window.

```
SELECT First([Shipped Date]) FROM Orders WHERE [Ship Country] = 'UK';
SELECT Last([Shipped Date]) FROM Orders WHERE [Ship Country] = 'UK';
```

The next example returns the shipping date of the first record in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition to limit the search to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= First([Shipped Date])
```

Fix Function

See **Int** Function, **Fix** Function

FontBold Property

Applies To Reports.

Description Determines the font style when you use the **Print** method on a report.

Setting The FontBold property settings are:

Setting	Description
–1	**Text is bold.**
0	(Default) Text is not bold.

Usage Design view: not available. Other views: read/write.

Remarks The appearance of bold text depends on your computer and printer. A font's appearance on screen and in print may differ. For example, if your printer doesn't support bold, Microsoft Access substitutes a similar font. Copy may appear lighter in print than it does on screen.

When you use the **Print** method, Microsoft Access prints the specified text on the report. If the FontBold property setting is **True** (–1), the text will appear bold. If the FontBold property setting is **False** (0), the text won't appear bold.

You can use the FontBold property in Access Basic code in the form of a function called from a macro attached to one of the event properties, such as OnFormat or OnPrint.

To use the FontBold property, first create a **Function** procedure that prints the desired text. For example, you could use the following procedure to print the current date on a report at the coordinates specified by the CurrentX and CurrentY property settings:

```
Function Print_Bold_Date
  Screen.ActiveReport.FontBold = True
  Screen.ActiveReport.Print Date()          ' Print date in bold.
End Function
```

Once you have created a **Function** procedure, you can create a macro that uses the RunCode action. Enter the name of the **Function** procedure for the Function Name argument.

See Also BorderColor Property; FillColor Property; FontItalic, FontUnderline Properties; FontName, FontSize Properties; FontWeight Property

FontItalic, FontUnderline Properties

Apply To

Reports. Tools and controls (combo box, command button, label, list box, text box, toggle button).

Description

- Reports—set the italic and underline attributes for text when the **Print** method is used.
- Tools and controls—set the italic and underline attributes for text in a control.

For reports, these properties can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

For tools and controls only, set this property using the Palette.

Setting

The FontItalic property settings are:

Setting	Description
Yes	*Text is italic.*
No	(Default) Text is normal.

The FontUnderline property settings are:

Setting	Description
Yes	Text is underlined.
No	(Default) Text is normal.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: not available for reports; read/write for tools and controls. Other views: read/write for reports; read-only for tools and controls.

See Also

BorderColor Property; FontName, FontSize Properties; FontWeight Property; **Print** Method

FontName, FontSize Properties

Apply To

Reports. Tools and controls (combo box, command button, label, list box, text box, toggle button).

Description

- Reports—specify the name and size of the font used for text with the **Print** method.
- Tools and controls—specify the name and size of the font used for text in a control.

For reports, these properties can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

For tools and controls only, set this property using the Palette.

Setting

For the FontName property, the font names that appear on the property sheet depend on the fonts installed on your system and for your printer. If you are using Microsoft Windows version 3.0, the default setting is Helv. If you are using Windows version 3.1, the default setting depends on the setting of the LayoutForPrint property.

LayoutForPrint	FontName default
No	MS Sans Serif
Yes	Arial

The FontSize property settings are:

Setting	Description
8	(Default for all except buttons) Text is 8-point type.
10	(Default for buttons) Text is 10-point type.
12	Text is 12-point type.
<Other sizes>	Text is the selected size.

Remarks

If you select a font that your system can't display or that isn't installed, Microsoft Windows substitutes a similar font.

Access Basic

Use a string expression to set the value of each property.

Design view: not available for reports; read/write for tools and controls. Other views: read/write for reports; read-only for tools and controls.

See Also

BorderColor Property; FontItalic, FontUnderline Properties; FontWeight Property, LayoutForPrint Property; **Print** Method

FontSize Property

See FontName, FontSize Properties

FontUnderline Property

See FontItalic, FontUnderline Properties

FontWeight Property

Applies To Tools and controls (combo box, command button, label, list box, text box, toggle button).

Description Determines the width of the line that Microsoft Windows uses to display and print characters in a control.

Setting The FontWeight property settings are Thin, Extra Light, Light, Normal, Medium, Semi-bold, Bold, Extra Bold, and Heavy. The following are examples of two of the settings.

Setting	Example
Normal	This is normal-weight type.
Bold	**This is bold-weight type**.

Remarks Font availability depends on your system and printer. If you select a font that your system can't display or that is not installed, Windows substitutes a similar font.

A font's appearance on screen and in print may differ. For example, a FontWeight of Thin may look identical to Normal on screen but appear lighter when printed.

Access Basic The property settings and their values are:

Setting	Value
Thin	100
Extra Light	200
Light	300
Normal	400
Medium	500

Setting	Value
Semi-bold	600
Bold	700
Extra Bold	800
Heavy	900

Design view: read/write. Other views: read-only.

See Also BorderColor Property; FontItalic, FontUnderline Properties; FontName, FontSize Properties

For...Next Statement

Description Repeats a group of instructions a specified number of times.

Syntax

For *counter* = *start* **To** *end* [**Step** *increment*]
[*statementblock*]
[**Exit For**]
[*statementblock*]
Next [*counter* [, *counter*]]

Remarks The **For** statement uses these arguments.

Argument	Description
counter	Numeric variable used as the loop counter. The variable cannot be an array element or a record element.
start	Initial value of the counter.
end	Final value of the counter.
increment	The amount the counter is changed each time through the loop. If you do not specify **Step**, *increment* defaults to one.
statementblock	Any number of statements or methods to be executed the specified number of times.

The **Step** value controls loop execution as follows.

When Step is	Loop executes if
Positive or 0	*counter* <= *end*
Negative	*counter* >= *end*

Once the loop has been entered and all the statements in the loop have executed, **Step** is added to *counter*. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute in the first place), or the loop is exited and execution continues with the statement following the **Next** statement.

Tip Changing the value of *counter* while inside a loop can make the program more difficult to read and debug.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its counter. The following construction is correct:

```
For I = 1 To 10
  For J = 1 To 10
    For K = 1 To 10
      ...
    Next K
  Next J
Next I
```

A **Next** statement with the form `Next K, J, I` is equivalent to the following sequence of statements:

```
Next K
Next J
Next I
```

Note If you omit the variable in a **Next** statement, the value of **Step** *increment* is added to the variable associated with the most recent **For** statement. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

See Also **Do...Loop** Statement, **While...Wend** Statement

Example This example puts five lines of the uppercase alphabet into `Msg` using two nested loops.

```
NL = Chr(13) & Chr(10)                    ' Define newline.
For Rep = 5 To 1 Step -1                  ' Set up five repetitions.
  For Indx = Asc("A") To Asc("Z")         ' Convert alpha to numbers.
    Msg = Msg & Chr(Indx)                 ' Append each letter to string.
  Next Indx
  Msg = Msg & NL                          ' Add newline for each rep.
Next Rep
```

ForceNewPage Property

Applies To Form and report sections (except page headers and page footers).

Description Determines whether a section starts printing on the current page or at the top of a new page.

Setting The ForceNewPage property settings are:

Setting	Description
No	(Default) The current section starts printing on the current page.
Before Section	The current section starts printing at the top of a new page.
After Section	The next section starts printing at the top of a new page.
Before & After	Combines the effects of Before Section and After Section.

Remarks Depending on the design of your form or report, you might want certain sections, such as the detail section, to start at the top of a page. However, if your design doesn't require a special layout, you can set the ForceNewPage property setting to No.

Access Basic The property settings and their values are:

Setting	Value
No	0
Before Section	1
After Section	2
Before & After	3

Design view: read/write. Other views: read-only.

See Also NewRowOrCol Property, OnFormat Property, Page Property

ForeColor Property

Applies To Reports. Tools and controls (combo box, command button, label, list box, text box, toggle button).

Description

- Reports—specifies the printing and drawing color for the **Print**, **Line**, and **Circle** methods.
- Tools and controls—specifies the color for text in a control.

For reports, this property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

Setting For tools and controls only, set this property using the Palette.

Remarks You can use this property to create special visual effects on forms or reports when you print with a color printer.

Setting the ForeColor property doesn't affect graphics or print output already drawn. For all other controls, the screen color changes in Design view.

Access Basic Use a numeric expression to set the value of this property. You can use the **RGB** or **QBColor** function in the expression.

Design view: not available for reports; read/write for tools and controls. Other views: read/write for reports; read-only for tools and controls.

The value of this property is a **Long** data type.

See Also BackStyle Property, DrawMode Property, DrawStyle Property, DrawWidth Property, OnFormat Property, OnPrint Property

Form, Report Properties

Apply To Forms. Reports.

Description Refer to the form or report itself or to the form or report associated with a subform or subreport.

Setting These properties have no settings.

Usage Design view: read-only. Other views: read-only.

Remarks You can use these properties in a macro to refer to the form or report that invokes the macro or in a function that refers to the active form or report. The following example uses the Form property in a function to refer to the form that contains the call to the function.

```
= MyFunction(Form![My Control])
```

When you use the Form property, it refers to the active form, and the name of the form isn't specified.

If the active form is `My Form`, the next example is equivalent to the preceding example.

```
Forms![My Form]![My Control]
```

When you use the **Forms** or **Reports** object, you must specify the name of the form or report. You can refer to any open form or report, not just the one that contains the expression.

The following examples use the Form and Report properties to refer to a control on a subform and subreport.

```
X = Forms![My Form]![My Subform].Form![My Control]
X = Reports![My Report]![My Subreport].Report![My Control]
```

Format, Format$ Functions

Description Format a number, date, time, or string according to instructions contained in a format expression.

Syntax **Format**[**$**](*expression*[, *fmt*])

Remarks **Format** returns a **Variant**; **Format$** returns a **String**.

The **Format**[**$**] function uses these arguments.

Argument	Description
expression	Numeric or string expression to be formatted.
fmt	Format expression—a string of display-format characters that specify how the expression is to be displayed. Different type format expressions (numeric, date/time, or string) cannot be mixed in a single *fmt* argument. You can also use the same named formats used by the Format property.

If *fmt* is omitted or is zero-length and *expression* is a numeric expression, **Format**[**$**] provides the same functionality as the **Str**[**$**] function by converting the numeric value to

the appropriate return data type. Note that positive numbers converted to strings using **Format[$]** lack the leading space reserved for displaying the sign of the value, whereas those converted using **Str[$]** retain the leading space.

You can use any of the following symbols to create a format expression for numbers.

Symbol	Meaning
Null string	Display the number with no formatting.
0	Digit placeholder. Display a digit or a zero. If there is a digit in the expression being formatted in the position where the 0 appears in the format string, display it; otherwise, display a zero in that position. If the number being formatted has fewer digits than there are zeros (on either side of the decimal) in the format expression, leading or trailing zeros are displayed. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, the number is rounded to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, the extra digits are displayed without modification.
#	Digit placeholder. Display a digit or nothing. If there is a digit in the expression being formatted in the position where the # appears in the format string, display it; otherwise, display nothing in that position. This symbol works like the 0 digit placeholder, except that leading and trailing zeros aren't displayed if the number has the same or fewer digits than there are # characters on either side of the decimal separator in the format expression.
. (period)	Decimal placeholder. The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only number signs to the left of this symbol, numbers smaller than 1 begin with a decimal separator. If you want a leading zero to always be displayed with fractional numbers, use 0 as the first digit placeholder to the left of the decimal separator instead. The actual character used as a decimal placeholder in the formatted output depends on the Number Format specified in the International section of the Microsoft Windows Control Panel. For some countries, a comma is used as the decimal separator.
%	Percentage placeholder. The expression is multiplied by 100. The percent character (%) is inserted in the position where it appears in the format string.

Symbol	Meaning
, (comma)	Thousand separator. The thousand separator separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Standard use of the thousand separator is specified if the format contains a comma surrounded by digit placeholders (0 or #). Two adjacent commas or a comma immediately to the left of the decimal separator (whether or not a decimal is specified) means "scale the number by dividing it by 1000, rounding as needed." You can scale large numbers using this technique. For example, you can use the format string "##0,," to represent 100 million as 100. Numbers smaller than 1 million are displayed as 0. The actual character used as the thousand separator in the formatted output depends on the Number Format specified in the International section of the Control Panel. For some countries, a period is used as the thousand separator.
E– E+ e– e+	Scientific format. If the format expression contains at least one digit placeholder (0 or #) to the right of E–, E+, e–, or e+, the number is displayed in scientific format and E or e is inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E– or e– to place a minus sign next to negative exponents. Use E+ or e+ to place a minus sign next to negative exponents and a plus sign next to positive exponents.
: (colon)	Time separator. The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator depends on the Time Format specified in the International section of the Control Panel.
/	Date separator. The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in the formatted output depends on Date Format specified in the International section of the Control Panel.
– + $ ( ) space	Display a literal character. To display a character other than one of those listed, precede it with a backslash (\) or enclose it in double quotation marks (" ").

Symbol	Meaning
\	Display the next character in the format string. Many characters in the format expression have a special meaning and can't be displayed as literal characters unless they are preceded by a backslash. The backslash itself isn't displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (\\). Examples of characters that can't be displayed as literal characters are the date- and time-formatting characters (a, c, d, h, m, n, p, q, s, t, w, y, /, and :), the numeric-formatting characters (#, 0, %, E, e, comma, and period), and the string-formatting characters (@, &, <, >, and !).
"*ABC*"	Display the string inside the double quotation marks. To include a string in *fmt* from within Access Basic, you must use **Chr**(34) to enclose the text (34 is the ANSI code for a double quotation mark). In property sheets, just type the double quotation character.
[*color*]	Display the string in a specified color. Valid colors are Black, Blue, Green, Cyan, Red, Magenta, Yellow, or White. Note that color can be specified only in Microsoft Access property sheets.
*	Display the next character as the fill character. Any empty space in a field is filled with the character following the asterisk.

Unless the *fmt* argument contains one of the named formats, a format expression for numbers can have one to four sections separated by semicolons.

If you use	The result is
One section only	The format applies to all values.
Two sections	The first section applies to positive values and zeros, the second to negative values.
Three sections	The first section applies to positive values, the second to negative values, and the third to zeros.
Four sections	The first section applies to positive values, the second to negative values, the third to zeros, and the fourth to **Null** values.

The following example has two sections: the first defines the format for positive values and zeros; the second section defines the format for negative values.

```
"$#,##0;($#,##0)"
```

If you include semicolons with nothing between them, the missing section is printed using the format of the positive value. For example, the following format displays positive and negative values using the format in the first section and displays "Zero" if the value is zero.

```
"$#,##0;;\Z\e\r\o"
```

Some sample formats for numbers are shown below. (These examples all assume that the International Country code is set for the United States.) The first column contains the format strings. The other columns contain the output that results if the formatted data has the value given in the column headings.

Format (*fmt*)	Positive 5	Negative 5	Decimal .5	Null
`Null string`	5	-5	0.5	
`0`	5	-5	1	
`0.00`	5.00	-5.00	0.50	
`#,##0`	5	-5	1	
`#,##0.00;;;Nil`	5.00	-5.00	0.50	Nil
`$#,##0;($#,##0)`	$5	($5)	$1	
`$#,##0.00;($#,##0.00)`	$5.00	($5.00)	$0.50	
`0%`	500%	-500%	50%	
`0.00%`	500.00%	-500.00%	50.00%	
`0.00E+00`	5.00E+00	-5.00E+00	5.00E-01	
`0.00E-00`	5.00E00	-5.00E00	5.00E-01	

You can format date and time serial numbers using date and time or number formats (because date/time serial numbers are stored as floating-point values.)

You can use any of the following symbols to create a format expression for dates and times.

Symbol	Meaning
c	Display the date as ddddd and display the time as t t t t t, in that order. Only date information is displayed if there is no fractional part to the date serial number; only time information is displayed if there is no integer portion.
d	Display the day as a number without a leading zero (1–31).
dd	Display the day as a number with a leading zero (01–31).
ddd	Display the day as an abbreviation (Sun–Sat).
dddd	Display the day as a full name (Sunday–Saturday).
ddddd	Display a date serial number as a complete date (including day, month, and year) formatted according to the Short Date setting in the International section of the Windows Control Panel. The default Short Date format is `m/d/yy`.
dddddd	Display a date serial number as a complete date (including day, month, and year) formatted according to the Long Date setting in the International section of the Control Panel. The default Long Date format is `mmmm dd, yyyy`.
w	Display the day of the week as a number (1 for Sunday through 7 for Saturday.)
ww	Display the week of the year as a number (1–53).
m	Display the month as a number without a leading zero (1–12). If m immediately follows h or hh, the minute rather than the month is displayed.
mm	Display the month as a number with a leading zero (01–12). If m immediately follows h or hh, the minute rather than the month is displayed.
mmm	Display the month as an abbreviation (Jan–Dec).
mmmm	Display the month as a full month name (January–December).
q	Display the quarter of the year as a number (1–4).
y	Display the day of the year as a number (1–366).
yy	Display the year as a two-digit number (00–99).
yyyy	Display the year as a four-digit number (100–9999).
h	Display the hour as a number without leading zeros (0–23).
hh	Display the hour as a number with leading zeros (00–23).

Symbol	Meaning
n	Display the minute as a number without leading zeros (0–59).
nn	Display the minute as a number with leading zeros (00–59).
s	Display the second as a number without leading zeros (0–59).
ss	Display the second as a number with leading zeros (00–59).
t t t t t	Display a time serial number as a complete time (including hour, minute, and second) formatted using the time separator defined by the Time Format in the International section of the Control Panel. A leading zero is displayed if the Leading Zero option is selected and the time is before 10:00 A.M. or P.M. The default time format is `h:nn:ss`.
AM/PM	Use the 12-hour clock and display an uppercase AM with any hour before noon; display an uppercase PM with any hour between noon and 11:59 PM.
am/pm	Use the 12-hour clock and display a lowercase AM with any hour before noon; display a lowercase PM with any hour between noon and 11:59 PM.
A/P	Use the 12-hour clock and display an uppercase A with any hour before noon; display an uppercase P with any hour between noon and 11:59 PM.
a/p	Use the 12-hour clock and display a lowercase A with any hour before noon; display a lowercase P with any hour between noon and 11:59 PM.
AMPM	Use the 12-hour clock and display the contents of the 1159 string (s1159) in the WIN.INI file with any hour before noon; display the contents of the 2359 string (s2359) with any hour between noon and 11:59 PM. AMPM can be either uppercase or lowercase, but the case of the string displayed matches the string as it exists in the WIN.INI file. The default format is AM/PM.

The following are examples of date and time formats.

Format	Display
`m/d/yy`	12/7/58
`d-mmmm-yy`	7-December-58
`d-mmmm`	7-December
`mmmm-yy`	December-58
`hh:mm AM/PM`	08:50 PM
`h:nn:ss a/p`	8:50:35 p
`h:mm`	20:50
`h:nn:ss`	20:50:35
`m/d/yy h:mm`	12/7/58 20:50

Strings can also be formatted with **Format[$]**. A format expression for strings can have one section or two sections separated by a semicolon.

If you use	The result is
One section only	The format applies to all string data.
Two sections	The first section applies to string data, the second to **Null** values and zero-length strings.

You can use any of the following symbols to create a format expression for strings.

Symbol	Meaning
@	Character placeholder. Display a character or a space. If there is a character in the string being formatted in the position where the @ appears in the format string, display it; otherwise, display a space in that position. Placeholders are filled from right to left unless there is an ! character in the format string. See below.
&	Character placeholder. Display a character or nothing. If there is a character in the string being formatted in the position where the & appears, display it; otherwise, display nothing. Placeholders are filled from right to left unless there is an ! character in the format string. See below.
<	Force lowercase. All characters are displayed in lowercase format.
>	Force uppercase. All characters are displayed in uppercase format.
!	Force placeholders to fill from left to right instead of right to left.

See Also **Str**, **Str$** Functions

Example This example uses **Format** to format date and time values according to the setting of entries in the International section of the WIN.INI file.

```
NL = Chr(13) & Chr(10)                          ' Define newline.
Msg = "Today's date is " & Format(Now, "ddddd") & "." & NL
Msg = Msg & "The current time is " & Format(Now, "ttttt")
Msg = Msg & "."
MsgBox Msg                                      ' Display date/time formatting.
```

Format Property

Applies To Table fields. Control (text box).

Description Specifies the format for displaying and printing numbers, dates, times, and text.

Setting The Format property has different settings for different data types. For more information about settings for specific data types, see one of the following topics:

- Format Property—Number and Currency Data Types
- Format Property—Date/Time Data Types
- Format Property—Yes/No Data Types
- Format Property—Text and Memo Data Types

Remarks Microsoft Access provides standard formats for Number, Date/Time, and Yes/No data types. You can select one of the standard formats or create a user-defined format using formatting symbols.

The standard formats depend on the country specified in the International section of the Microsoft Windows Control Panel. Microsoft Access displays formats appropriate for the country selected. For example, with the Currency format set for the United States, 1234.56 appears as $1,234.56; but when the format is set for the United Kingdom, it appears as £1,234.56.

User-defined formats for fields containing numbers (Number, Date/Time, Currency, Yes/No) can have one to four sections separated by semicolons. User-defined formats for fields containing text (Text, Memo) can have one or two sections. You can use one or more sections. If a section has no format, Microsoft Access uses the format from the first section.

You can't mix user-defined formatting symbols for Number data types with Date/Time symbols or with Text format symbols. For more technical information about user-defined formatting symbols, see **Format, Format$** Functions.

The sections for user-defined number formats have the following meanings.

Section	Meaning
First	Positive numbers
Second	Negative numbers
Third	Zero values
Fourth	Null or empty values

For example, the standard Currency format has this pattern: "$#,##0.00;($#,##0.00)[Red]." It uses a different format for positive numbers (determined by the first section) than for negative numbers (determined by the second section).

The sections for user-defined text formats have the following meanings.

Section	Meaning
First	Fields with text
Second	Fields with no text

You might have a text box in which you want the word None to appear if there is no value in the field. You can enter "@;None" as a user-defined format. The @ symbol in the first section tells Microsoft Access to display the text from the field. The second section of the user-defined format tells Microsoft Access to display the word None when there is no value in the field.

You can use the following symbols in user-defined formats for any data type.

Symbol	Meaning
(Space)	Display spaces as literal characters.
"*ABC*"	Display anything inside quotation marks as literal characters.
!	Force left alignment instead of right alignment.
*	Fill available space with the following character.
\	Display the next character as a literal character. You can use quotation marks around literal characters.
[*color*]	Display in the color named between the brackets. Available colors: Black, Blue, Green, Cyan, Red, Magenta, Yellow, White.

Note When you create a user-defined format, the alignment defaults to right-aligned. Use an exclamation point to force left alignment.

Access Basic

The Format property can have text values for user-defined formats, such as "dd-mmm-yy," and can have the following numerical values for standard formats.

Setting	Value
General Number	0
Currency	1
Fixed	2
Standard	3
Percent	4
Scientific	5
General Date	6
Long Date	7
Medium Date	8
Short Date	9

Setting	Value
Long Time	10
Medium Time	11
Short Time	12
Yes/No	13
True/False	14
On/Off	15

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also FieldSize Property; **Format**, **Format$** Functions

Format Property—Date/Time Data Types

Setting The following table shows the Format property settings for Date/Time data types. For more information, see the example.

Setting	Description
General Date	(Default) If the value is date only, no time is displayed; if the value is time only, no date is displayed. Examples: 4/3/93 05:34 PM, 4/3/93, and 05:34 PM.
Long Date	Same as the Long Date setting in the International section of the Microsoft Windows Control Panel. Example: Saturday, April 4, 1993.
Medium Date	Example: 03-Apr-93.
Short Date	Same as the Short Date setting in the International section of the Windows Control Panel. Example: 4/3/93.
Long Time	Same as the Time setting in the International section of the Windows Control Panel. Example: 5:34:23 PM.
Medium Time	Example: 05:34 PM.
Short Time	Example: 17:34.

You can also create user-defined date and time formats using the following codes.

Setting	Description
: (colon)	Time separator. Separators are set in the International section of the Windows Control Panel.
/	Date separator.
c	Same as the standard General Date format.
d	Day of the month in one or two numerical digits, as needed (1–31).
dd	Day of the month in two numerical digits (01–31).
ddd	First three letters of the weekday (Sun–Sat).
dddd	Full name of the weekday (Sunday–Saturday).
ddddd	Same as the standard Short Date format.
dddddd	Same as the standard Long Date format.
w	Day of the week (1–7).
ww	Week of the year (1–53).
m	Month of the year in one or two numerical digits as needed (1–12).
mm	Month of the year in two numerical digits (01–12).
mmm	First three letters of the month (Jan–Dec).
mmmm	Full name of the month (January–December).
q	Date displayed as the quarter of the year (1–4).
y	Number of the day of the year (1–366).
yy	Last two digits of the year (01–99).
yyyy	Full year (0100–9999).
h	Hour in one or two digits, as needed (0–23).
hh	Hour in two digits (00–23).
n	Minute in one or two digits, as needed (0–59).
nn	Minute in two digits (00–59).
s	Second in one or two digits, as needed (0–59).
ss	Second in two digits (00–59).
ttttt	Same as the standard Long Time format.

Setting	Description
AM/PM	Twelve-hour clock with the uppercase letters AM or PM, as appropriate.
am/pm	Twelve-hour clock with the lowercase letters AM or PM, as appropriate.
A/P	Twelve-hour clock with the uppercase letter A or P, as appropriate.
a/p	Twelve-hour clock with the lowercase letter A or P, as appropriate.
AMPM	Twelve-hour clock with the appropriate forenoon/afternoon designator as defined in the International section of the Windows Control Panel.

Example

The following are examples of Date/Time formats.

Setting	Display
ddd", " mmm d", "yyyy	Mon, Jun 2, 1996
mmmm dd", "yyyy	June 02, 1996
"This is week number "ww	This is week number 22
"Today is "dddd	Today is Tuesday

Access Basic

The Format property can have text values for user-defined formats, such as "dd-mmm-yy," and can have the following numerical values for standard formats.

Setting	Value
General Number	0
Currency	1
Fixed	2
Standard	3
Percent	4
Scientific	5
General Date	6
Long Date	7
Medium Date	8
Short Date	9

Setting	Value
Long Time	10
Medium Time	11
Short Time	12
Yes/No	13
True/False	14
On/Off	15

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also **Format**, **Format$** Functions

Format Property—Number and Currency Data Types

Setting

The following table shows the Format property settings for numbers. For more information, see the example.

Setting	Description
General Number	(Default) Display number as entered.
Currency	Use thousand separator; display negative numbers in red and enclose them in parentheses; DecimalPlaces property setting is 2.
Fixed	Display at least one digit; DecimalPlaces property setting is 2.
Standard	Use thousand separator; DecimalPlaces property setting is 2.
Percent	Multiply value by 100; append a percent sign; DecimalPlaces property setting is 2.
Scientific	Use standard scientific notation.

You can also create user-defined number formats using the following codes.

Setting	Description
. (period)	Decimal separator. Separators are set in the International section of the Microsoft Windows Control Panel.
, (comma)	Thousand separator.
0	Digit placeholder. Display a digit or 0.
#	Digit placeholder. Display a digit or nothing.
$	Display the literal character $.

Setting	Description
%	Percentage. Value is multiplied by 100 and the percent sign is appended.
E– or e–	Scientific notation with a minus sign next to negative exponents. It must be used with other symbols, as in 0.00E–00.
E+ or e+	Scientific notation with a minus sign next to negative exponents and a plus sign next to positive exponents. It must be used with other symbols, as in 0.00E+00.

Remarks

Use the DecimalPlaces property to display a different number of decimal places than the Format property settings allow.

Example

The following are examples of the standard number formats.

Setting	Unformatted	Formatted
General Number	3456.789 –3456.789 $213.21	3456.789 –3456.789 $213.21
Currency	3456.789 –3456.789	$3,456.79 ($3,456.79)
Fixed	3456.789 –3456.789 3.56645	3456.79 –3456.79 3.57
Standard	3456.789	3,456.79
Percent	3 0.45	300% 45%
Scientific	3456.789 –3456.789	3.46E+03 –3.46E+03

Access Basic

The Format property can have text values for user-defined formats, such as "dd-mmm-yy," and can have the following numerical values for standard formats.

Setting	Value
General Number	0
Currency	1
Fixed	2
Standard	3
Percent	4
Scientific	5

Setting	Value
General Date	6
Long Date	7
Medium Date	8
Short Date	9
Long Time	10
Medium Time	11
Short Time	12
Yes/No	13
True/False	14
On/Off	15

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also DecimalPlaces Property; FieldSize Property; **Format**, **Format$** Functions

Format Property—Text and Memo Data Types

Setting You can use the following symbols to create user-defined formats.

Setting	Description
@	Text character (either a character or a space) required
&	Text character not required
<	Force all characters to lowercase
>	Force all characters to uppercase

Example The following are examples of Text and Memo formats.

Setting	Data	Display
@@@-@@-@@@@	465-04-3799	465-0-4--3799
	465043799	465-04-3799
@@@@@@@@@@	465-04-3799	465-04-3799
	465043799	465043799

Setting	Data	Display
>	davolio	DAVOLIO
	DAVOLIO	DAVOLIO
	Davolio	DAVOLIO
<	davolio	davolio
	DAVOLIO	davolio
	Davolio	davolio

Access Basic

The Format property can have text values for user-defined formats, such as "dd-mmm-yy," and can have the following numerical values for standard formats.

Setting	Value
General Number	0
Currency	1
Fixed	2
Standard	3
Percent	4
Scientific	5
General Date	6
Long Date	7
Medium Date	8
Short Date	9
Long Time	10
Medium Time	11
Short Time	12
Yes/No	13
True/False	14
On/Off	15

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also

DecimalPlaces Property; FieldSize Property; **Format**, **Format$** Functions

Format Property—Yes/No Data Types

Setting

For Yes/No data types, the Format property settings are:

Setting	Description
Yes/No	(Default) No = 0, Yes = –1
True/False	False = 0, True = –1
On/Off	Off = 0, On = –1

User-defined formats for Yes/No data types can have one to four sections:

- The first section is not used for these data types but does require a semicolon placeholder.
- The second section is for values equal to –1 (Yes).
- The third section is for values equal to 0 (No).
- The fourth section is not used for these data types.

The following example displays the word Always in blue text for Yes, True, or On and the word Never in red text for No, False, or Off:

```
";Always[Blue];Never[Red]"
```

Access Basic

The Format property can have text values for user-defined formats, such as "dd-mmm-yy," and can have the following numerical values for standard formats.

Setting	Value
General Number	0
Currency	1
Fixed	2
Standard	3
Percent	4
Scientific	5
General Date	6
Long Date	7
Medium Date	8
Short Date	9
Long Time	10
Medium Time	11

Setting	Value
Short Time	12
Yes/No	13
True/False	14
On/Off	15

Design view: read/write. Other views: read-only. Table fields aren't available.

See Also **Format**, **Format$** Functions

FormatCount Property

Applies To Reports.

Description Indicates the number of times the OnFormat property setting is evaluated for the current line.

Setting The FormatCount property setting is a **Long** value.

Usage Design view: read-only. Other views: read-only.

Remarks Microsoft Access increments the FormatCount property each time the OnFormat property setting is evaluated for the current line. As the next line is formatted, Microsoft Access resets FormatCount to 0.

Under some circumstances, a line might be formatted more than once. For example, you might design a report to print address labels that each include three lines: name, address, and city. Microsoft Access formats and prints the labels one page at a time. If the last label on a page won't fit (there might be room for only two lines), Microsoft Access prints it on the next page. In this case, the FormatCount property setting for the lines is 2 because each line is formatted twice, once for each page.

You might also print a report that contains order information. As each line is printed, you can keep a running total of the order amounts. The following example uses the FormatCount property to make sure each order is counted only once.

```
If Reports![My Report].FormatCount = 1 Then
  RunningTotal = RunningTotal + OrderAmount
End If
```

See Also PrintCount Property

FormName Property

Applies To Forms. Reports.

Description Identifies a form or report.

Setting The value of this property is a string indicating the name of the form or report as it appears in the Database window.

To set the FormName property of a new form or report, choose the Save command from the File menu in Design view and enter a valid name. The default name is the object type plus a unique integer, such as Form3 or Report1. To change the name of a form or report, choose the Rename command from the File menu in the Database window.

Usage Design view: read-only. Other views: read-only.

Remarks Microsoft Access uses this property's setting to identify an object, as in the following example:

```
X = Reports![MyReport].FormName
```

See Also ControlName Property

Forms Object

Description You use the **Forms** object to refer to a particular form. For example, you can use the **Forms** object to read or set the value of a form control.

The **Forms** object has no methods and only one property: Count. The following example shows how you can use the Count property to calculate the number of open forms.

```
NumberForms = Forms.Count
```

You can refer to a form in three ways:

- **Forms**![My Form]
- **Forms**("My Form")
- **Forms**(*form number*)

If you use **Forms**(*form number*), notice that the forms are zero-based; that is, the first form is **Forms**(0), the second form is **Forms**(1), and so on.

You can use the **Forms** object in Access Basic and in expressions.

See Also ActiveControl Property, Count Property, **Debug** Object, **Reports** Object, **Screen** Object

Example This example reads the value of the Last Name and First Name controls from the Employees form in the NWIND.MDB database.

```
EmpName = Forms![Employees]![Last Name] & ", " & Forms![Employees]!
➥ [First Name]
```

The next example uses the **Forms** object in an expression. If the Ship Country field of the Orders form in the NWIND.MDB database is **Null**, the expression returns an empty string; otherwise, it returns the field's contents.

```
IIf(IsNull(Forms![Orders]![Ship Country]), "", Forms![Orders]![Ship Country])
```

The next example displays the name of each form that is currently open.

```
Sub ListForms
  Dim X As Integer
  NumberForms = Forms.Count                ' Count number of forms.
  Debug.Print "There are"; NumberForms; "open forms:"
  For X = 0 To NumberForms - 1
    Debug.Print Forms(X).Caption           ' Print form name.
  Next X
End Sub
```

FreeFile Function

Description Returns the next valid unused file number.

Syntax **FreeFile** [( )]

Remarks If **FreeFile** is used in anything except Access Basic code in a Microsoft Access module, the empty parentheses must be included.

Use **FreeFile** when you need to supply a file number and you want to ensure that the file number is not already in use.

See Also **Open** Statement

Example In this example, the **FreeFile** function returns the next available file number.

```
FileNum = FreeFile                         ' Get unused file number.
Open "TESTFILE" For Output As FileNum      ' Create test file.
Close FileNum                              ' Close file.
```

Function Statement

Description Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

[Static] [Private] Function *functionname* [(*argumentlist*)] [**As** *type*]
[*statementblock*]
[*functionname* = *expression*]
[**Exit Function**]
[*statementblock*]
[*functionname* = *expression*]
End Function

Remarks All executable code must be in **Function** or **Sub** procedures. You can't define a **Function** procedure inside another **Function** or **Sub** procedure.

The **Function** statement uses these arguments.

Argument	Description
Static	Indicates that the **Function** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Function** procedure, even if they are used in the procedure.
Private	Indicates that the **Function** procedure is accessible only to other procedures in the module in which it exists. No other procedure in any other module has access.
functionname	Name of the function. **Function** procedure names follow the same rules that constrain the names of other variables and can include a type-declaration character. Because **Function** names are recognized by all procedures in all modules, *functionname* cannot be the same as any other globally recognized name in the program. Such names include the names of procedures (in Access Basic or a dynamic-link library [DLL]), **Global** variables, and **Global** constants. To avoid a name conflict, you can use the **Private** reserved word to make the **Function** procedure local to the module in which it appears, but access to it will not be available from any procedure in any other module. Even if you use the **Private** reserved word, *functionname* still cannot be the same as any other module-level variable, constant, or procedure name in the same module.

Argument	Description
	Note that the type of the name determines the data type returned by the **Function** procedure. To specify the return data type of the **Function** procedure, either use an **As** clause following *argumentlist* or include a type-declaration character in the **Function** name. For example, to create a **Function** procedure that returns a string, you can include a dollar sign in the name, use an **As String** clause, or give it a name defined as a string name with a **DefStr** statement.
argumentlist	List of variables, representing arguments, that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas. Unless identified with the **ByVal** reserved word, individual arguments are passed by reference, so changing an argument's value inside the **Function** procedure changes its value in the calling procedure. If an argument passed to a function is an expression, it is treated as if it used the **ByVal** reserved word: no part of the expression is altered by the function.
As *type*	Reserved word used to declare the data type of the value returned by the **Function** procedure; *type* can be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String**, or **Variant**.
statementblock	Any group of statements that are to be executed within the body of the **Function** procedure.
expression	Return value of the function. A **Function** procedure returns a value by assigning that value to the function *functionname*. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *functionname*, the procedure returns a default value: A numeric function returns 0, a **String** function returns a zero-length string (""), and a **Variant** function returns Empty.
Exit Function	Causes an immediate exit from a **Function** procedure. Program execution continues with the statement following the statement that called the **Function** procedure. Any number of **Exit Function** statements can appear anywhere in a **Function** procedure.

The argument *argumentlist* has the following syntax:

[**ByVal**]*variable*[( )] [**As** *type*] [, [**ByVal**]*variable*[( )] [**As** *type*]] . . .

It uses these arguments.

Argument	Description
ByVal	Indicates that the argument is passed by value rather than by reference. The **ByVal** reserved word cannot be used with a variable of a user-defined type or object.
variable	Name of the variable representing the argument. For array variables, use the parentheses but omit the number of dimensions.
As *type*	Declares the data type of *variable*. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (variable-length strings only), **Variant**, a user-defined type, or an object data type (but no arrays of objects). Use a separate **As** *type* clause for each argument.

Function and **End Function** mark the beginning and end of a **Function** procedure.

Like a **Sub** procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a **Sub** procedure, a **Function** procedure can be used in an expression in the same manner as any intrinsic function, such as **Sqr**, **Cos**, or **Chr**.

You can call a **Function** procedure by using the function name, followed by the argument list in parentheses, in an expression. If the function has no arguments, you still must include the parentheses.

Caution **Function** procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow. The **Static** reserved word is usually not used with recursive **Function** procedures.

To return a value from a function, assign the value to the function name. For example, in a function named `BinarySearch`, you might assign the value of the constant **False** to the name to indicate that the value was not found.

```
Function BinarySearch(...)
...
  ' Value not found. Return a value of False.
  If lower > upper Then
    BinarySearch = False
    Exit Function
  End If
...
End Function
```

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Other variables used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Caution A procedure can use a variable that is not explicitly declared in the procedure, but a name conflict can occur if anything you have defined in the Declarations section has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, a **Global** or module-level constant or variable, or an object, Access Basic assumes that your procedure is referring to that module-level name. Explicitly declare variables to avoid this kind of conflict. You can use an **Option Explicit** statement to force explicit declaration of variables.

Avoid using I/O statements in a **Function** procedure called from another I/O statement; doing so can cause unpredictable results.

Access Basic may rearrange arithmetic expressions to increase internal efficiency. Avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

See Also **Dim** Statement, **Global** Statement, **Option Explicit** Statement, **Static** Statement, **Sub** Statement

Example In this example, the **Function** statement declares the name of a **Function** procedure. **Exit Function** is used to exit the procedure before the **End Function** statement is encountered.

```
Function SquareRoot (X As Double) As Double
  Select Case Sgn(X)                    ' Evaluate sign of argument.
    Case 1                              ' OK if positive.
      SquareRoot = Sqr(X)
      Exit Function
    Case 0                              ' Notify user if 0.
      Msg = "The value passed was 0."
    Case -1                             ' Notify user if negative.
      Msg = "Illegal value."
  End Select
  MsgBox Msg                            ' Display message.
End Function
```

FV Function

Description Returns the future value of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **FV**(*rate*, *nper*, *pmt*, *pv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **FV** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pmt	Payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

The arguments *rate* and *nper* must be calculated using payment periods expressed in the same units. For example, if *rate* is calculated using months, *nper* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **IPmt** Function, **NPer** Function, **Pmt** Function, **PPmt** Function, **PV** Function, **Rate** Function

Example

This example returns the future value of an investment given the percentage rate that accrues per period (`APR / 12`), the total number of payments (`TotPmts`), the payment (`Payment`), the current value of the investment (`PVal`), and a number that indicates whether the payment is made at the beginning or end of the payment period (`PayType`). Note that because `Pmt` represents cash paid out, it's a negative number.

```
Const ENDPERIOD = 0, BEGINPERIOD = 1      ' When payments are made.
Const MB_YESNO = 4                        ' Define Yes/No buttons.
Const ID_NO = 7                           ' Define No as a response.
Fmt = "###,###,##0.00"                    ' Define money format.
Payment = InputBox("How much do you plan to save each month?")
APR = InputBox("What is the expected interest annual percentage rate?")
If APR > 1 Then APR = APR / 100           ' Ensure proper form.
TotPmts = InputBox("For how many months do you expect to save?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
PVal = InputBox("How much is in this savings account now?")
FVal = FV(APR / 12, TotPmts, -Payment, -PVal, PayType)
MsgBox "Your savings will be worth " & Format(FVal, Fmt) & "."
```

Get Statement

Description

Reads from a disk file into a variable.

Syntax

Get [#] *filenumber*, [*recordnumber*], *variablename*

Remarks

The **Get** statement uses the following arguments.

Argument	Description
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.
recordnumber	The largest possible *recordnumber* is $2^{31} - 1$, or 2,147,483,647. For files opened in **Random** mode (with the **Random** reserved word), *recordnumber* is the number of the record to be read. For files opened in **Binary** mode (with the **Binary** reserved word), *recordnumber* is the byte position at which reading starts. The first byte in a file is at position 1, the second byte is at position 2, and so on. If you omit *recordnumber*, the next record or byte (the one following the last **Get** or **Put** statement or the one pointed to by the last **Seek** function) is read from. If you omit *recordnumber*, you must still include the commas. For example: `Get #4,,FileBuffer`

Argument	Description
variablename	Name of the variable used to receive input from the file. Any variable can be used except object variables and array variables (variables that describe an entire array). However, you can use a variable that describes a single element of an array.

For files opened in **Random** mode, the following rules apply:

- If the length of the data being read is less than the length specified in the **Len** clause of the **Open** statement, **Get** still reads subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be determined with any certainty, it is generally a good idea to have record length match the length of the data being written.
- If the variable being read is a variable-length string, **Get** reads a 2-byte descriptor containing string length and then the variable. The record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual length of the string.
- If the variable being read is a numeric **Variant** (**VarType** 0–7), **Get** reads 2 bytes identifying the **VarType** of the **Variant** and then the variable. For example, when reading a **Variant** of **VarType** 3, **Get** reads 6 bytes: 2 bytes identifying the **Variant** as **VarType** 3 (**Long**) and 4 bytes containing the **Long** data. The record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual number of bytes required to store the variable.
- If the variable being read is a **String Variant** (**VarType** 8), **Get** reads 2 bytes identifying the **VarType**, then 2 bytes indicating the length of the string, and then the string data. The record length specified by the **Len** clause in the **Open** statement must be at least 4 bytes greater than the actual length of the string.
- If the variable being read is any other type of variable (not a variable-length string or a **Variant**), **Get** reads only the variable data. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the length of the data being written.
- Elements of user-defined types are read using **Get** as if each were being read individually except that there is no padding between elements. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the sum of all the bytes required to write the individual elements.

For files opened in **Binary** mode, all of the **Random** rules apply except that:

- In **Binary** mode, the **Len** clause in the **Open** statement has no effect. **Get** reads all variables from disk contiguously, that is, with no padding between records.
- **Get** reads variable-length strings that aren't elements of user-defined types without expecting the 2-byte length descriptor. The number of bytes read equals the number of characters in the string. For example, the following statements read 10 bytes from file number 1:

```
VarString$ = String$(10," ")
Get #1, , VarString$
```

See Also **Open** Statement, **Put** Statement, **Type** Statement

Example In this example, the **Get** statement reads data from a disk file into a variable.

```
Dim ClientName As String * 20              ' Declare a fixed-length variable.
Dim Indx As Long
Open "CLIENTS.DAT" For Random As #1 Len = Len(ClientName$)
Max = LOF(1) / Len(ClientName)             ' Determine number of entries.
For Indx = 1 To Max                        ' Loop to cycle through entries.
  Get #1, Indx, ClientName                 ' Read from file.
  MsgBox ClientName                        ' Show entry in a message box.
Next Indx
Close #1                                   ' Close file.
```

GetChunk Method

Description Returns all or a portion of a Memo or an OLE Object field in a specified recordset.

Syntax *stringvar* = *recordset* ! *field*.**GetChunk**(*offset*, *numbytes*)

Remarks The **GetChunk** method uses the following arguments.

Argument	Description
stringvar	Name of a string variable or **Variant** that will receive the data from the field named by *field*
recordset	Name of an open recordset object
field	Name of a Memo or an OLE Object field
offset	Numeric expression that is the number of bytes to skip before copying begins
numbytes	Numeric expression that is the number of bytes you want to return

If *offset* is 0, **GetChunk** begins copying from the first byte of the field.

Assign the bytes returned by **GetChunk** to *stringvar*. Because a string variable is limited in size, *numbytes* can't be larger than approximately 64K. If the field is larger than 64K, use **GetChunk** to return a portion at a time. You can use the **AppendChunk** method to reassemble the pieces. If *numbytes* is larger than 64K, an error occurs even if *stringvar* is a **Variant**.

If *numbytes* is larger than the number of bytes in the field, the actual number of bytes of data in the field are returned. After assigning the results of **GetChunk** to a string variable, you can use the **Len** function to determine the number of bytes returned.

If the value of *offset* or *numbytes* is less than 0, **GetChunk** uses the two's complement of the value.

See Also **AppendChunk** Method, **FieldSize** Method, **Len** Function

Example This example uses the **GetChunk** method to return consecutive 16K chunks of the Notes field in the Employees table of the NWIND.MDB database. It copies the chunks to the string array `NoteArray( )`, where they can be changed. **AppendChunk** then copies `NoteArray( )` back to `Notes`.

```
Option Base (1) ' This line should appear in the Declarations section.

Sub Get_Notes
  Const ChunkSize = 16384                              ' Set size of chunk.
  Dim NumChunks As Integer, TotalSize As Long, X As Integer
  Dim MyDB As Database, MyTable As Table
  Set MyDB = CurrentDB()
  Set MyTable = MyDB.OpenTable("Employees")            ' Open Table.
  TotalSize = MyTable!Notes.FieldSize()                ' Get field size.
                                                       ' How many chunks?
  NumChunks = TotalSize \ ChunkSize - (TotalSize Mod ChunkSize <> 0)
  ReDim NoteArray(NumChunks) As String * ChunkSize
                                                       ' Get current Notes field.
  For X = 1 To NumChunks
    NoteArray(X) = MyTable!Notes.GetChunk((X - 1) * ChunkSize, ChunkSize)
  Next X
  ...                                                  ' Make changes.
  MyTable.Edit                                         ' Enable editing.
  MyTable!Notes = ""                                   ' Initialize Notes field.
  For X = 1 To NumChunks
    MyTable!Notes.AppendChunk(NoteArray(X))    ' Replace with edited notes.
  Next X
  MyTable.Update                                       ' Save changes.
  MyTable.Close                                        ' Close Table.
End Sub
```

Global Statement

Description Used at the module level to declare **Global** variables and allocate storage space.

Syntax **Global** *variablename*[([*subscripts*])] [**As** *type*] [, *variablename*[([*subscripts*])] [**As** *type*]] . . .

Remarks The **Global** statement uses these arguments.

Argument	Description
variablename	Name of a variable.
subscripts	Dimensions of an array. You can declare multiple dimensions. The syntax of *subscripts* is described below.
As *type*	Reserved word used to declare the data type of a variable. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Variant**, a user-defined type, or an object data type (but no arrays of objects). Use a separate **As** *type* clause for each variable being defined.

The argument *subscripts* in the **Global** statement has the following syntax:

[*lower* **To**]*upper*[, [*lower* **To**] *upper*] . . .

The **To** reserved word provides a way to indicate both the lower and upper bounds of an array variable's subscripts. The following statements are equivalent if there is no **Option Base** statement:

```
Global A(8,3)
Global A(0 To 8, 0 To 3)
Global A(8, 0 To 3)
```

Array subscripts can be negative. **To** can be used to specify any range of subscripts between –32,768 and 32,767, inclusive:

```
Global A(-4 To 10)
Global B(-99 To -5, -3 To 0)
```

The maximum number of array dimensions allowed in a **Global** statement is 60.

You can calculate the amount of memory used by a numeric array by multiplying the number of elements in the array times the number of bytes required by the data type of the array. For example, an **Integer** array, which requires 2 bytes per element, can contain twice as many elements as a **Long** array, which requires 4 bytes per element. An **Integer** array can contain four times as many elements as a **Double** array, which requires 8 bytes per element. **String** arrays are limited to less than 64K bytes because some small storage overhead is required for each array.

If you use a subscript that is greater than the specified maximum or smaller than the specified minimum, an error occurs and a message is displayed, unless it is trapped in error-handling code. An error also occurs if the size of the array (in terms of bytes of memory used) exceeds the allowable limits described above.

Use the **Global** statement to declare variables that are global (available to all procedures in all modules). You can also use the **Global** statement to declare the data type of a variable. For example, the following statement declares the variable as an **Integer**:

```
Global NumberOfEmployees As Integer
```

You can also use the **Global** statement with empty parentheses to declare dynamic arrays. After declaring a dynamic array, use the **ReDim** statement within a procedure level to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size has already been declared, an error occurs.

Variables are initialized at compile time. Numeric variables are initialized to 0, **Variant** variables to Empty. Variable-length strings are initialized as zero-length strings, and fixed-length strings are filled with zeros. The fields of user-defined type variables are initialized as if they were separate variables.

See Also **Const** Statement, **Dim** Statement, **Option Base** Statement, **ReDim** Statement, **Static** Statement, **Type** Statement

Example The **Global** statement declares two variables in a module. The first is a global dynamic-array variable; the second is a global **Integer** variable.

```
Global TestArray() As Integer          ' Declare global dynamic-array variable.
Global Size As Integer                 ' Declare global Integer variable.
```

GoSub...Return Statements

Description Branch to and return from a subroutine within a procedure.

Syntax

GoSub {*linelabel* | *linenumber*}

. . .

linelabel: or *linenumber*

. . .

Return

Remarks The arguments *linelabel* and *linenumber* are labels that mark the first line of the subroutine.

The argument *linelabel* must begin with an alphabetic character, must end with a colon, must be 40 characters or less, and must not be an Access Basic reserved word. Each

linelabel must be unique in the module in which it is used. Line labels are not case-sensitive and may begin in any column as long as they are the first nonblank characters on the line.

The argument *linenumber* can be any number of 40 characters or less, can contain only decimal digits (0–9), and must not end with a colon. Each *linenumber* must be unique in the module in which it appears. Line numbers may begin in any column as long as they are the first nonblank characters. If *linenumber* is greater than 65,529, the **Erl** function cannot properly report the line number where an error has occurred.

GoSub and **Return** can be used anywhere in a procedure, but the **GoSub** and corresponding **Return** must occur within the same procedure. A subroutine can contain more than one **Return** statement, but the first **Return** statement encountered causes program flow to branch back to the statement immediately following the most recent **GoSub** statement.

Note Entering or exiting **Sub** blocks with **GoSub...Return** is not supported.

Caution Subroutines that call themselves (recursive subroutines) can easily run out of stack space and halt your program unexpectedly.

See Also **End** Statement, **GoTo** Statement, **On...GoSub** Statement, **On...GoTo** Statement, **Sub** Statement

Example This example uses **GoSub** to call a subroutine from within a **Sub** procedure. Notice that the **GoTo** branch avoids the subroutine. To avoid this kind of branching, turn subroutines into **Sub** procedures.

```
Sub GoDemo()
  Num = InputBox("Type a number.")
  GoSub Routine                          ' Branch to subroutine.
  GoTo Nextpart                          ' Branch around subroutine.
Routine:                                 ' Start of subroutine.
  Num = Num / 2
  Return                                 ' Return from subroutine.
Nextpart:                                ' Continuation of program.
  Msgbox "Half of your number is " & Str(Num)
End Sub
```

GoTo Statement

Description Branches unconditionally to a specified line within a procedure.

Syntax **GoTo** { *linelabel* | *linenumber*}

Remarks The arguments *linelabel* and *linenumber* indicate the line to execute next.

The argument *linelabel* must begin with an alphabetic character, must end with a colon, must be 40 characters or less, and must not be an Access Basic reserved word. Each *linelabel* must be unique in the module in which it is used. Line labels are not case-sensitive and may begin in any column as long as they are the first nonblank characters on the line.

The argument *linenumber* can be any number of 40 characters or less, can contain only decimal digits (0–9), and must not end with a colon. Each *linenumber* must be unique in the module in which it appears. Line numbers may begin in any column as long as they are the first nonblank characters. If *linenumber* is greater than 65,529, the **Erl** function cannot properly report the line number where an error has occurred.

GoTo can branch only to lines within the procedure in which it appears.

Note Programs with many **GoTo** statements can be difficult to read and debug. Use structured control statements (**Do...Loop**, **For...Next**, **If...Then...Else**, **Select Case**) whenever possible.

See Also **Do...Loop** Statement, **For...Next** Statement, **GoSub...Return** Statements, **If...Then...Else** Statement, **Select Case** Statement

Example See **GoSub...Return** Statements

GoToControl Action

Description Moves the focus to the specified field or control on the open form, form datasheet, table datasheet, or query dynaset.

Action argument	Description
Control Name	The name of the field or control where you want the focus. Required argument.

Remarks This action moves the focus to the specified field or control in the current record.

Enter only the name of the field or control in the Control Name argument, not the full syntax (such as **Forms**!Products![Product ID]).

You can use this action when you want a particular field or control to have the focus. This field or control can then be used for comparisons or FindRecord actions. You can also use this action to navigate in a form according to certain conditions. For example, if the user enters Male in a Sex field on a health insurance form, the focus can automatically skip the Pregnancy Coverage field and move to the next field.

Tip You can use the GoToControl action to move to a subform, which is a type of control. You can then use the GoToRecord action to move to a particular record in the subform. You can also move to a control on a subform by using the GoToControl action to move first to the subform and then to the control on the subform.

See Also GoToPage Action, GoToRecord Action, SelectObject Action

Access Basic

Syntax

DoCmd GoToControl *controlname*

Argument	Description
controlname	A string expression that is the name of a control on the active form or datasheet

Remarks

Use only the name of the control for the *controlname* argument, not the full syntax.

You can also use a variable dimensioned as a **Control** data type for the *controlname* argument:

```
Dim MyControl As Control
Set MyControl = Forms!Form1!Field3
DoCmd GoToControl MyControl.ControlName
```

Example

This example uses the GoToControl action to move the focus to the `Employee ID` field.

```
DoCmd GoToControl "Employee ID"
```

GoToPage Action

Description Moves the focus in the active form to the first field on a specified page.

Action argument	Description
Page Number	The number of the page to which you want to move the focus. Required argument.
Right	The horizontal position of the page's upper-left corner, measured from the left edge of its containing window.
Down	The vertical position of the page's upper-left corner, measured from the top edge of its containing window.

Note The Right and Down measurements are in inches or centimeters, depending on the units set in the International section of the Microsoft Windows Control Panel.

Remarks Use this action to select the first field (as defined by the form's tab order) on the specified page. Use the GoToControl action to move to a particular field or other control on the form.

You can use this action if you have created a form with page breaks that group related information. For example, you might have an Employees form with personal information on one page, office information on another page, and sales information on a third page. You can use the GoToPage action to move to the desired page.

You can use the Right and Down arguments for forms with pages larger than the Microsoft Access window. Use the Page Number argument to move to the desired page, and then use the Right and Down arguments to display the part of the page you want to see. Microsoft Access displays the part of the page whose upper-left corner is offset the specified distance from the upper-left corner of the page.

See Also GoToControl Action, GoToRecord Action, SelectObject Action

Access Basic

Syntax

DoCmd GoToPage *pagenumber* [, *right*] [, *down*]

Argument	Description
pagenumber	A numeric expression that is a valid page number for the active form
right	A numeric expression that is a valid horizontal offset for the page
down	A numeric expression that is a valid vertical offset for the page

Remarks

The units for the *right* and *down* arguments are twips.

If you specify the *down* argument and leave the *right* argument blank, you must include the *right* argument's comma. If you leave a trailing argument blank, don't use a comma following the last argument you specify.

Example

This example uses the GoToPage action to move the focus to the position specified by the horizontal and vertical offsets on the second page of the active form.

```
DoCmd GoToPage 2, 1440, 567
```

GoToRecord Action

Description Makes the specified record the current record in an open table, form, or query dynaset.

Action argument	Description
Object Type	The type of object that contains the record you want to make current. You can select Table, Query, or Form from this argument's drop-down list in the Action Arguments section of the Macro window. Leave this argument blank to select the active object.
Object Name	The name of the object that contains the record you want to make current. The drop-down list shows all objects in the current database of the type selected with the Object Type argument. If you leave Object Type blank, leave this argument blank also.
Record	The record to make current. Select Previous, Next, First, Last, Go To, or New from the drop-down list. The default is Next.
Offset	An integer or expression (preceded by an equal sign) that evaluates to an integer. This argument specifies the record to make current. You can use Offset in two ways: When the Record argument is Next or Previous, Microsoft Access moves the Offset number of records toward the end or beginning of the records. When the Record argument is Go To, Microsoft Access moves to the record whose record number equals the Offset number. The record number is shown in the record number box at the bottom of the screen.

Note If you select First, Last, or New for Record, Microsoft Access ignores any Offset number. If you enter an Offset number that is too large, Microsoft Access displays an error message. You can't enter negative Offset numbers.

Remarks

If the focus is in a particular field in a record, this action leaves it in the same field in the new record.

Use the New setting for the Record argument to move to the empty record at the end of a form or table so that you can enter new data.

This action is similar to choosing the Go To command from the Records menu. The First, Last, Next, Previous, and New subcommands of the Go To command have the same effect on the selected object as the First, Last, Next, Previous, and New settings for the Record argument. You can also move to records using the navigation buttons at the bottom of the window.

See Also

GoToControl Action, GoToPage Action, SelectObject Action

Access Basic

Syntax

DoCmd GoToRecord [*objecttype*, *objectname*] [, *record*] [, *offset*]

Argument	Description
objecttype	One of the following intrinsic constants: A_TABLE A_QUERY A_FORM
objectname	A string expression that is the valid name of an object of the type selected with the *objecttype* argument.
record	One of the following intrinsic constants: A_PREVIOUS A_NEXT A_FIRST A_LAST A_GOTO A_NEWREC If you leave this argument blank, the default (A_NEXT) is assumed.
offset	A numeric expression that represents the number of records to move forward or backward if you specify A_NEXT or A_PREVIOUS for the *record* argument, or the record to move to if you specify A_GOTO for the *record* argument. The expression must result in a valid record number.

Remarks

If you leave the *objecttype* and *objectname* arguments blank, the active object is assumed.

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example uses the GoToRecord action to make the seventh record in the form `My Form` current.

```
DoCmd GoToRecord A_FORM, "My Form", A_GOTO, 7
```

GridX, GridY Properties

Apply To Forms. Reports.

Description Specify the horizontal and vertical units of the alignment grid in a form's or report's Design view.

Setting Enter an integer for the number of subdivisions per unit of measurement, typically inches or centimeters. The default setting is 10 for GridX (horizontal) and 12 for GridY (vertical).

Remarks The GridX and GridY properties provide control over the placement and alignment of objects on a form or report. You can adjust the grid for greater or lesser precision. To see the grid, choose the Grid command from the View menu.

Access Basic Use a numeric expression to set the value of each property.

Design view: read/write. Other views: read-only.

Height, Width Properties

Apply To

- Height—form and report sections; tools and controls (all except page breaks).
- Width—forms; reports; tools and controls (all except page breaks).

Description Specify the height and width of an object.

Setting Enter a number for the desired height or width in the current unit of measurement. To use a unit other than the default, include a measurement indicator, such as "cm" or "in."

The Height and Width properties are also set when you draw or size a control or when you size a window in a form's or report's Design view.

Remarks

The height and width of forms and reports are measured from the inside of their borders. The height and width of controls are measured from the center of their borders so that controls with different border widths align correctly.

The right margin of a form or report is determined by the left margin and the Width property setting for the form or report. For example, if a report's left margin is 1 and the report's Width property setting is 6.5, the right margin will be 1.

The bottom margin of a form or report is determined by the Top and Height property settings for the bottom section on the form or report. For example, if a form has a page header section, a detail section, and a page footer section, the bottom margin is determined by the page footer section. If the section's Top property is set to 3 and its Height property is set to 2, the bottom margin will be 5 units from the top of the form.

Note To set the left and top margins, use the Left and Top properties.

Access Basic

Use a numeric expression to set the value of each property. Tool properties aren't available from Access Basic.

Design view: read/write. Other views: read-only.

Although the Width property is read-only for forms and reports, you can resize forms and reports using the MoveSize action.

Note Values are expressed in twips (1 twip equals 1/1440 of an inch, or about 1/567 of a centimeter).

See Also

BorderWidth Property; Left, Top Properties; MoveSize Action

HelpContextID, HelpFile Properties

Apply To

- HelpContextID—forms and reports; tools and controls (bound object frame, check box, combo box, command button, list box, object frame [linked], option button, option group, text box, toggle button).
- HelpFile—forms and reports.

Description

- HelpContextID—sets the identifying number for a topic in the custom Help file specified by the HelpFile property setting.
- HelpFile—specifies the name of a custom Help file for the current form.

Setting

- HelpContextID—enter an integer between 0 and 2,147,483,647. The default is 0.
- HelpFile—enter the name of a custom Microsoft Windows Help file created with the Microsoft Windows Help Compiler. This file must be in the same directory as the database or in a directory listed in the path in the AUTOEXEC file.

Remarks

You can create a custom Help file to document forms or applications you create with Microsoft Access.

When you press the F1 key in Form view, Microsoft Access calls the Windows Help application, loads the Help file specified by the form's HelpFile property, and requests the Help topic specified by the HelpContextID property. If a control's HelpContextID is not 0, Microsoft Access uses that setting to identify the Help topic. If the control's HelpContextID is 0 (the default), then Microsoft Access uses the form's HelpContextID to identify the Help topic. If you press F1 when not in Form view, or if HelpContextID for both the control and the form is 0, a Microsoft Access Help topic is displayed.

Note To create a custom Help file, you need the Microsoft Windows Help Compiler and an RTF text editor, such as Microsoft Word for Windows.

Access Basic

Use a string expression or a numeric expression to set the value of the HelpContextID property. Use a string expression to set the value of the HelpFile property.

Design view: read/write. Other views: read-only.

HelpFile Property

See HelpContextID, HelpFile Properties

Hex, Hex$ Functions

Description

Return a string that represents the hexadecimal value of a decimal argument.

Syntax

Hex[$](*number*)

Remarks

Hex returns a **Variant**; **Hex$** returns a **String**.

The argument *number* can be any numeric expression. It is rounded to the nearest whole number before being evaluated. An error occurs if *number* is a **Variant** of **VarType** 1 (**Null**).

If the argument is an **Integer** or a **Variant** of **VarType** 2 (**Integer**), up to four hexadecimal characters are returned; if the argument is a **Long**, or a **Variant** of **VarType** 3 (**Long**), up to eight hexadecimal characters are returned.

You can represent hexadecimal numbers directly by preceding numbers in the proper range with `&H`. For example, `&H10` represents decimal 16 in hexadecimal notation.

See Also **Oct**, **Oct$** Functions

Example This example returns the hexadecimal representation of the numeric variable `Num`.

```
= Hex(Num)
```

HideDuplicates Property

Applies To Controls (check box,* combo box, list box, option button,* option group, text box, toggle button*) on a report.

* Except when the control is in an option group.

Description Specifies whether a control is visible when its data is the same as its data in the previous record.

Setting The HideDuplicates property settings are:

Setting	Description
Yes	If the data in the control is the same as its data in the previous record, the control is hidden.
No	(Default) The control is visible regardless of the data.

Remarks Use this property to make a report easier to read. This is generally used in a report's detail section. For example, you might have a list of suppliers and their products. If a supplier provides several products, you can show the supplier's name once instead of placing the name beside every product name.

Duplicates aren't suppressed in the first record of a group or in the first record on a page.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1 (nonzero)
No	0

Design view: read/write. Other views: read-only.

Hour Function

Description Returns an integer between 0 and 23, inclusive, that represents the hour of the day corresponding to the time provided as an argument.

Syntax **Hour**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Minute** Function, **Month** Function, **Now** Function, **Second** Function, **Weekday** Function, **Year** Function

Example See **Minute** Function

Hourglass Action

Description Changes the mouse pointer to an image of an hourglass while a macro is running.

Action argument	Description
Hourglass On	Select Yes to display the hourglass. Select No to display the normal mouse pointer. The default is Yes.

Remarks Use this action to provide a visual indication that the macro is running. This is especially useful when a macro action or the macro itself takes a long time to run.

Often, you might use this action if you've turned echo off using the Echo action. When echo is off, Microsoft Access suspends screen updates until the macro is finished.

Microsoft Access automatically sets Hourglass On back to No when the macro finishes running.

See Also Echo Action

Access Basic

Syntax

DoCmd Hourglass *hourglasson*

Argument	Description
hourglasson	Use the reserved word **True** (–1) to display the hourglass. Use the reserved word **False** (0) to display the normal mouse pointer.

Example

This example uses the Hourglass action to display an hourglass while the macro is running.

```
DoCmd Hourglass True
```

hWnd Property

Applies To

Forms. Reports.

Description

The value of this property is a handle (an **Integer**) that Microsoft Windows assigns to the current window.

Setting

Design view: not available. Other views: read-only.

Remarks

Use this property when making calls to Windows routines from Access Basic. Many Windows functions require the hWnd of the current window as one of their arguments. One such function is the Windows IsZoomed function, which you can use to ensure a window is maximized.

```
' The following line should appear in the Declarations section.
Declare Function IsZoomed Lib "USER" (ByVal hWnd As Integer) As Integer

If Not IsZoomed(Screen.ActiveForm.hWnd) Then
  DoCmd Maximize
End If
```

Caution

Because the value of this property can change while a program is running, don't store the hWnd value in a variable. Code that calls Windows functions to draw graphics on reports may not work in future versions of Microsoft Access.

See Also

Declare Statement, **DoCmd** Statement, Maximize Action

If...Then...Else Statement

Description Allows conditional execution, based on the evaluation of an expression.

Syntax 1 **If** *condition* **Then** *thenpart* [**Else** *elsepart*]

Syntax 2

If *condition1* **Then**
    [*statementblock-1*]
[**ElseIf** *condition2* **Then**
    [*statementblock-2*]]
[**Else**
    [*statementblock-n*]]
End If

Remarks

Syntax 1

The single-line form is often useful for short, simple conditional tests. It has the following parts.

Part	Description
condition	One of two types of expressions: A numeric or string expression that evaluates true (nonzero) or false (0 and **Null**). A single expression of the form **If TypeOf** *object* **Is** *objecttype*. The *object* is a control variable and *objecttype* can be any one of the following: BoundObjectFrame, CheckBox, ComboBox, CommandButton, Graph, Label, Line, ListBox, ObjectFrame (for embedded and linked unbound object frames), OptionButton, OptionGroup, PageBreak, Rectangle, SubForm, SubReport, TextBox, or ToggleButton.
thenpart, elsepart	Statements or branches performed when *condition* is true (*thenpart*) or false (*elsepart*).

The *thenpart* and the *elsepart* fields both have this syntax:

{*statements* | [**GoTo**] *linenumber* | **GoTo** *linelabel* }

The *thenpart* and *elsepart* syntax has these parts.

Part	Description
statements	One or more Access Basic statements, separated by colons
linenumber	A valid program line number
linelabel	A valid program line label

Note that **GoTo** is optional with a line number but is required with a line label.

The *thenpart* is executed if *condition* is true or if *object* is of the type specified by *objecttype*; if *condition* is false or *object* is not of the specified *objecttype*, then *elsepart* is executed. If the **Else** clause is not present, control passes to the next statement in the program.

You can have multiple statements with a condition, but they must be on the same line and separated by colons, as in the following statement:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

Syntax 2

The block form of **If...Then...Else** provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug. It has the following parts.

Part	Description
condition1, condition2	The same conditions as used in the single-line form
statementblock-1 to -n	One or more Access Basic statements on one or more lines

In executing a block **If**, Access Basic tests *condition1*, the first numeric expression or **TypeOf** expression. If the expression is true or if *object* is of the type specified by *objecttype*, the statements following **Then** are executed. If the first expression is false or if *object* is not of the type specified by *objecttype,* Access Basic begins evaluating each **ElseIf** condition in turn. When Access Basic finds a true condition or an *object* that is of the type specified by *objecttype,* the statements following the associated **Then** are executed. If none of the **ElseIf** conditions is true or no *object* is of the type specified in its associated *objecttype,* the statements following the **Else** are executed. After executing the statements following **Then** or **Else**, the program continues with the statement following **End If**.

The **Else** and **ElseIf** blocks are both optional. You can have as many **ElseIf** clauses as you would like in a block **If**, but none can appear after an **Else** clause. Any of the statement blocks can contain nested block **If** statements.

Access Basic looks at what appears after the **Then** reserved word to determine whether or not an **If** statement is a block **If**. If anything other than a comment appears after **Then**, the statement is treated as a single-line **If** statement.

A block **If** statement must be the first statement on a line. The **Else**, **ElseIf**, and **End If** parts of the statement can have only a line number or line label in front of them. The block must end with an **End If** statement.

Standard numeric and string expressions cannot be evaluated in the same conditional expression with expressions that use the form **If TypeOf** *object* **Is** *objecttype*. Use a separate **ElseIf** block for objects. For example:

```
If a > 1 And a <= 100 Then
  ...
ElseIf TypeOf controlVar Is ListBox Then
  ...
End If
```

Note The **Option Compare** statement affects string comparison.

Tip **Select Case** may be more useful when evaluating a single expression that has several possible actions.

See Also **Select Case** Statement

Example This example illustrates the various forms of the **If...Then...Else** syntax.

```
X = InputBox("Enter a number greater than 0 and less than 1000.")
If X < 10 Then
  Y = 1                                  ' 1 digit.
ElseIf X < 100 Then
  Y = 2                                  ' 2 digits.
Else
  Y = 3
End If
If Y > 1 Then Unit = " digits." Else Unit = " digit."
MsgBox "The number you entered has " & Y & Unit
```

IIf Function

Description Returns one of two arguments depending on the evaluation of an expression.

Syntax **IIf**(*expr*, *truepart*, *falsepart*)

Remarks The **IIf** function uses the following arguments.

Argument	Description
expr	Expression you want to evaluate
truepart	Value or expression returned if *expr* is **True** (–1)
falsepart	Value or expression returned if *expr* is **False** (0)

You might use **IIf** (immediate if) to evaluate an expression and return either of two other values. For example, you could use the **IIf** function to inspect a field on a form and determine whether it was **Null**. If it was, you might have the function return an empty string; otherwise, it would return the field's contents. The following example checks the Ship Country field in the Orders table in the NWIND.MDB database.

```
IIf(IsNull(Forms!Orders![Ship Country]), "", Forms!Orders![Ship Country])
```

IIf is most useful within expressions that are not part of Access Basic, such as those on a form or in a report. In Access Basic, the more full-featured **If...Then...Else** statement offers more versatility.

On a form, **IIf** evaluates either *truepart* or *falsepart*, whichever is appropriate. In Access Basic, however, **IIf** evaluates both *truepart* and *falsepart*, even though it returns only one of them. Because of this, you should watch for undesirable side effects. For example, if evaluating *falsepart* results in a `Division by zero` error, an error occurs even if *expr* is **True**.

See Also **Choose** Function, **DLookup** Function, **If...Then...Else** Statement

Example This example evaluates the `Order Amount` field and returns the word `Large` if the amount is greater than 1000; otherwise, it returns the word `Small`.

```
= IIf([Order Amount] > 1000, "Large", "Small")
```

In the next example, **IIf** returns an empty string if the `Ship Country` field is **Null**; otherwise, it returns the field's contents.

```
= IIf(IsNull([Ship Country]), "", [Ship Country])
```

The next example shows how you can use **IIf** to return a string containing an employee's first and last name, along with a middle initial. If there is no initial, the expression returns the first and last names separated by a single space. If there is a middle initial, however, it includes the middle initial and a period.

```
= [First Name] & IIf(IsNull([Initial]), "", " " & [Initial] & ".") & " " &
➥ [Last Name]
```

Imp Operator

Description Used to perform a logical implication on two expressions.

Syntax *result* = *expr1* **Imp** *expr2*

Remarks The following table illustrates how *result* is determined.

If *expr1* is	And *expr2* is	*result* is
true (nonzero)	true	**True** (–1)
true	false (0)	**False** (0)
true	**Null**	**Null**
false	true	**True**
false	false	**True**
false	**Null**	**True**
Null	true	**True**
Null	false	**Null**
Null	**Null**	**Null**

The **Imp** operator performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

If bit in *expr1* is	And bit in *expr2* is	*result* is
0	0	1
0	1	1
1	0	0
1	1	1

Bit-wise comparisons can be performed only in Access Basic code.

See Also **And** Operator, Comparison Operators, **Eqv** Operator, **Not** Operator, Operator Precedence, **Or** Operator, **Xor** Operator

Example

This example prints a message that depends on the value of variables `A`, `B`, and `C`, assuming that no variable is a **Null**. If `A` = 10, `B` = 8, and `C` = 6, both expressions evaluate **True**. Because both are **True**, the **Imp** expression is also **True**.

```
If A > B Imp B > C Then
  Debug.Print "The left expression implies the right expression."
Else
  Debug.Print "The left expression doesn't imply the right expression."
End If
```

In Operator

Description

Determines whether the value of an expression is equal to any of several values in a specified list.

Syntax

expr [**Not**] **In** (*value1, value2, . . .*)

Remarks

The **In** operator uses the following arguments.

Argument	Description
expr	Expression identifying the field that contains the data you want to evaluate
value1, value2	Expression or list of expressions against which you want to evaluate *expr*

If *expr* is found in the list of values, the **In** operator returns **True** (–1); otherwise, it returns **False** (0). You can include the **Not** logical operator to evaluate the opposite condition (that is, whether *expr* is not in the list of values).

You might use **In** to determine whether the value of a control is in a set of specified regions. The following example shows how you can tell if an order is shipped to a local region. If the `Ship Region` is `Avon`, `Glos`, or `Som`, the **IIf** function returns `"Local"`. Otherwise, it returns `"Nonlocal"`.

```
=IIf([Ship Region] In ('Avon','Glos','Som'), "Local", "Nonlocal")
```

If *expr* or any of the entries in the list of values is **Null**, **In** returns a **Null**.

You can use the **In** operator in a query expression or in a calculated control on a form or report.

See Also

Between...And Operator, **Like** Operator, **Not** Operator

Example

The following example uses the Orders table in the NWIND.MDB database to create a query that includes orders shipped to Avon, Gloucester, and Somerset. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT * FROM Orders WHERE [Ship Region] In ('Avon','Glos','Som');
```

The next example sets the text for a control on a form based on the Orders table in the NWIND.MDB database. Depending on which region an order is shipped to, it sets the text for the control. You can enter this expression in a calculated control on the form.

```
=IIf([Ship Region] In ('Avon','Glos','Som'), "2-Day Delivery",
➥ "5-Day Delivery")
```

Index Property

Applies To

Tables.

Description

Sets the current index that Microsoft Access uses to order records in a table and in recordsets created from that table. This property can be used only in Access Basic code.

Setting

The Index property setting is a string that is the name of an index (PrimaryKey or Index1–5) or the name of the field that was used to define the index.

Usage

Read/write.

Remarks

A **Table** isn't ordered when you first open it. Set the Index property to the name of an index to set the order to the primary key or to some other key.

The specified index must already be defined and cannot be created at run time. If you set the Index property to an index that doesn't exist, or if the index isn't set when you use the **Seek** method, an error occurs.

You can use the **ListIndexes** method to obtain a list of existing indexes in a table.

The records in a **Table** can be ordered only according to the indexes defined for it. To sort the **Table** records in some other order, create a **Dynaset** or **Snapshot** that has a different sort order.

The Index property applies only to **Tables**. To specify the sort order for **Dynasets** and **Snapshots**, use the Sort property.

See Also

Indexed Property, Index1...Index5 Properties, **ListIndexes** Method

Example This example sets the index for `MyTable` (based on the Orders table of the NWIND.MDB database) to the primary key (`PrimaryKey` is the name of the primary key). Next, it locates the record with a matching key field value of 10,050. Notice that the current index must be set before certain operations (such as the **Seek** method) can be performed.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Orders")        ' Open Table.
MyTable.Index = "PrimaryKey"                  ' Set current index.
MyTable.Seek "=", 10050                       ' Locate record.
...
MyTable.Close                                 ' Close Table.
```

Index1...Index5 Properties

Apply To Tables.

Description Set a multiple-field index.

Setting Enter the names of two or more fields separated by semicolons.

Remarks The Index1...Index5 properties are used to find and sort records using at least two fields in a table. For example, you can create an index on the last name, first name, and middle initial fields in an Employees table. To create single-field indexes, use the Indexed property.

A table can have up to five multiple-field indexes. They are created when you save the table design and are automatically updated when you change or add records. You can add or delete indexes at any time in a table's Design view.

Note You can't index Memo, Yes/No, and OLE Object data type fields.

Access Basic These properties are available with the **ListIndexes** method.

See Also Indexed Property, PrimaryKey Property

Indexed Property

Applies To Table fields.

Description Sets a single-field index.

Setting The Indexed property settings are:

Setting	Description
No	(Default) No index.
Yes (Duplicates OK)	The index allows duplicates.
Yes (No Duplicates)	The index doesn't allow duplicates.

Remarks Use the Indexed property to find and sort records using a single table field. The field can hold either unique or non-unique values. For example, you can create an index on a driver's license field in an Employees table in which each license number is unique. Or you can create an index on a name field in which some names may be duplicates.

You can create as many indexes as you need. The indexes are created when you save the table design and are automatically updated when you change or add records. You can add or delete indexes at any time in a table's Design view.

If the primary key for a table is a single field, Microsoft Access automatically sets the Indexed property for that field to Yes (No Duplicates).

Note You can't index Memo, Yes/No, and OLE Object data type fields. To create multiple-field indexes, use the Index1...Index5 properties.

Access Basic This property is available with the **ListIndexes** method.

See Also Index1...Index5 Properties, PrimaryKey Property

Input, Input$ Functions

Description Return characters read from a sequential file.

Syntax **Input**[$](*n*, [#]*filenumber*)

Remarks **Input** returns a **Variant**; **Input$** returns a **String**.

The **Input**[$] function uses these arguments.

Argument	Description
n	Number of characters (bytes) to read from the file. It must be less than 65,535.
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.

The **Input**[$] function can be used only with files opened in **Input** or **Binary** mode.

Unlike the **Input #** statement, **Input**[$] returns all of the characters it reads, including commas, carriage returns, linefeeds, quotation marks, and leading spaces.

See Also **Input #** Statement

Example In this example, the **Input** function reads one character at a time from a file. When an ANSI character code 10 (linefeed) is encountered, the entire line of text is displayed in a message box.

```
Open "TESTFILE" For Input As #1          ' Open file.
Do While Not EOF(1)
  Char = Input(1, #1)                    ' Get one character.
  If Char <> Chr(10) Then                ' If not linefeed, add to line.
    TextData = TextData & Char
  Else                                   ' If linefeed,
    MsgBox TextData                      ' display line.
    TextData = ""                        ' Clear line.
  End If
Loop                                     ' Loop if not at end of file.
Close # 1                                ' Close file.
```

Input # Statement

Description Reads data from a sequential file and assigns the data to variables.

Syntax **Input #** *filenumber*, *variablelist*

Remarks The **Input** statement uses these arguments.

Argument	Description
filenumber	Number used to open the file with the **Open** statement. It can be any numeric expression that evaluates to the number of an open file. Note that the number sign (#) preceding *filenumber* is not optional.
variablelist	Comma-delimited list of the variables that are assigned values read from the file.

A variable in *variablelist* can't be an array variable but can be a variable that describes an element of an array. A variable also can't be a user-defined type variable but can be an element. All other variables are allowed, except object variables.

Data items in a file must appear in the same order as the variables in *variablelist*. Be sure to match data in the file with variables of the proper data type: string data into **String** or **Variant** variables, numeric data into numeric or **Variant** variables, and uniquely **Variant** data (dates and **Null** values) into **Variant** variables. Failure to match the data with a variable of the correct data type may produce unpredictable results.

When a variable in *variablelist* is numeric, the first nonspace character encountered is assumed to be the start of a number. The end of the number is assumed when a space, a comma, or the end of a line is encountered. Blank lines are input as zero. If the data is not valid numeric data, zero is the value assigned to the variable.

When a variable in *variablelist* is a **String**, the first nonspace character encountered is assumed to be the start of a character string. If that first character is a double quotation mark ("), it is ignored, but all characters following it (including spaces and commas) up to the next double quotation mark are input into the **String** variable. For strings not delimited by double quotation marks, the end of a string is assumed when a comma, a space, or the end of a line is encountered. Blank lines are input as zero-length strings.

When the *variablelist* variable is a **Variant**, the first nonspace character is assumed to be the start of the data. The following rules determine how the actual data is treated:

- If the input data is a valid number, the **Variant** is set to an appropriate numeric **VarType**.
- If the input data consists of no data, only a delimiting comma or a blank line, the **Variant** is assigned **VarType** 0 (Empty).
- If the input data is the literal `#NULL#`, the **Variant** is assigned **VarType** 1 (**Null**).

- If the input data is a date literal (a date of the form #yyyy-mm-dd hh:mm:ss#), the **Variant** is assigned **VarType** 7 (Date). Note that when either the date portion or the time portion is present, the other portion is optional.
- If the input data is not a valid number, is not Empty or **Null**, and is not a valid date, it is input as a **String** and the **Variant** is assigned **VarType** 8.

If the end of file is reached while a data item is being input, the input is terminated and an error occurs.

See Also **Input, Input$** Functions; **Write #** Statement

Example In this example, the **Input #** statement reads a value from a file into a variable.

```
Dim FileData As Variant
Open "MYDATA.DAT" For Input As #1
Do While Not EOF(1)
  Input #1, FileData
  MsgBox FileData
Loop
Close #1
```

```
' Open to read file.
' Continue until end of file.
' Read line of data.
' Display data read from file.

' Close file.
```

InputBox, InputBox$ Functions

Description Display a prompt in a dialog box, wait for the user to input text or choose a button, and return the contents of the edit box.

Syntax **InputBox**[$](*prompt* [, [*title*] [, [*default*][, *xpos*, *ypos*]]])

Remarks **InputBox** returns a **Variant**; **InputBox$** returns a **String**.

The **InputBox**[$] function uses the following arguments.

Argument	Description
prompt	String expression displayed as the message in the dialog box
title	String expression displayed in the title bar of the dialog box
default	String expression displayed in the edit box as the default response if no other input is provided

Argument	Description
xpos	Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen
ypos	Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen

The maximum length of *prompt* is approximately 255 characters and depends on the width of the characters used. If *prompt* will consist of more than one line, be sure to include a carriage return (ANSI character 13) and a linefeed (ANSI character 10) between each line.

If you omit the argument *title*, nothing is placed in the title bar. If you omit *default*, the edit box is displayed empty. If you omit *xpos*, you must also omit *ypos*. If *xpos* and *ypos* are omitted, the dialog box is horizontally centered and vertically positioned approximately one third of the way down the screen. If you omit either *title* or *default* or both but include *xpos* and *ypos*, you must still include the commas that separate the arguments.

If the user chooses the OK button or presses ENTER, the **InputBox[$]** function returns whatever is in the edit box. If the user chooses the Cancel button, the function returns a zero-length string ("").

See Also **MsgBox** Function, **MsgBox** Statement

Example This example prompts the user with a message and returns the user-entered value.

```
Msg = "Enter a value from 1 to 3."          ' Set prompt.
Title = "InputBox Demo"                     ' Set title.
Defvalue = "1"                              ' Set default return value.
Answer = InputBox(Msg, Title, Defvalue)     ' Get user input.
```

InStr Function

Description Returns the position of the first occurrence of one string within another string.

Syntax 1 **InStr(** [*start*,] *strexpr1*, *strexpr2*)

Syntax 2 **InStr(***start*, *strexpr1*, *strexpr2*, *compare*)

Remarks The **InStr** function uses these arguments.

Argument	Description
start	Numeric expression that sets the starting position for each search; *start* must be between 1 and approximately 65,535. If *start* is omitted, the search of *strexpr1* begins at the first character position. The *start* argument is not optional if the *compare* argument is specified.
strexpr1	String expression being searched.
strexpr2	String expression being sought.
compare	Specifies the string-comparison method. The argument *compare* must be 0, 1, 2, or the value of the CollatingOrder field returned by the **ListFields** method, and *start* must also be specified. If *compare* is 0, string comparison is case-sensitive. If *compare* is 1, string comparison is not case-sensitive. If *compare* is 2, the string-comparison method is **Database**, which uses the New Database Sort Order. If *compare* is omitted, **InStr** uses the string-comparison method set by the **Option Compare** statement. However, if the module doesn't contain an **Option Compare** statement, the default method is **Binary**.

If *strexpr2* is found within *strexpr1*, **InStr** returns the position at which the match was found. If *strexpr2* is zero-length, *start* is returned. If *start* is greater than *strexpr2*, *strexpr1* is zero-length, or *strexpr2* can't be found, **InStr** returns 0.

If either string expression is **Null**, the function returns a **Null**. If *start* or *compare* is **Null**, an error occurs.

See Also **Option Compare** Statement

Example In this example, **InStr** returns 13 (the position of the letter M in the uppercase string) unless **Option Compare Binary** overrides **Option Compare Database** (which Microsoft Access inserts in each new module). If it is overridden, **InStr** returns 39 (the position of the letter M in the lowercase string).

```
=InStr(1, "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz","m")
```

Int Function, Fix Function

Description Return the integer portion of a number.

Syntax **Int**(*number*)

Fix(*number*)

Remarks The argument *number* can be any valid numeric expression. Both **Int** and **Fix** remove the fractional part of *number* and return the resulting integer value.

The data type of the return value is the same as that of the *number* argument. However, if *number* is a **Variant** of **VarType** 8 (**String**) that can be converted to a number, the return type will be a **Variant** of **VarType** 5 (**Double**). If the numeric expression results in a **Null**, **Int** and **Fix** return a **Null**.

The difference between **Int** and **Fix** is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts –8.4 to –9, and **Fix** converts –8.4 to –8.

Fix(*number*) is equivalent to:

Sgn(*number*) * **Int**(**Abs**(*number*))

See Also **Abs** Function, **Atn** Function, **Cos** Function, Data Type Conversion Functions, Derived Math Functions, **Exp** Function, **Log** Function, **Rnd** Function, **Sgn** Function, **Sin** Function, **Sqr** Function, **Tan** Function

Example This example illustrates the difference between **Int** and **Fix**. **Int** returns –100, and **Fix** returns –99.

```
= Int(-99.8)
= Fix(-99.8)
```

Integer Data Type

Description **Integer** variables are stored as 16-bit numbers (2 bytes) ranging in value from –32,768 to 32,767. The type-declaration character for an **Integer** is **%** (ANSI character 37).

In addition to representing numbers in this range, **Integers** can represent Boolean values (values that can have one of only two states). For Boolean values, –1 represents true and 0 represents false. You can also use the Access Basic reserved words **True** and **False**, which return –1 and 0, respectively.

You can also use **Integers** to represent enumerated values. An enumerated value can contain a finite set of unique whole numbers, each of which has special meaning in the context in which it is used. Enumerated values provide a convenient way to select among a known number of choices.

See Also Access Basic Data Types, **Long** Data Type

IPmt Function

Description Returns the interest payment for a given period of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **IPmt**(*rate*, *per*, *nper*, *pv*, *fv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **IPmt** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
per	Payment period in the range 1 through *nper*.
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance, is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

The arguments *rate* and *nper* must be calculated using payment periods expressed in the same units. For example, if *rate* is calculated using months, *nper* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **FV** Function, **NPer** Function, **Pmt** Function, **PPmt** Function, **PV** Function, **Rate** Function

Example This example calculates how much of a payment is interest when all the payments are of equal value. Given are the interest percentage rate per period (`APR / 12`), the payment period for which the interest portion is desired (`Period`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Const ENDPERIOD = 0, BEGINPERIOD = 1      ' When payments are made.
Const MB_YESNO = 4                        ' Define Yes/No buttons.
Const ID_NO = 7                           ' Define No as a response.
FVal = 0                                  ' Usually 0 for a loan.
Fmt = "###,###,##0.00"                    ' Define money format.
PVal = InputBox("How much do you want to borrow?")
APR = InputBox("What is the annual percentage rate of your loan?")
If APR > 1 Then APR = APR / 100           ' Ensure proper form.
TotPmts = InputBox("How many monthly payments?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
For Period = 1 To TotPmts                 ' Total all interest.
  IntPmt = IPmt(APR / 12, Period, TotPmts, -PVal, FVal, PayType)
  TotInt = TotInt + IntPmt
Next Period
Msg = "You'll pay a total of " & Format(TotInt, Fmt)
Msg = Msg & " in interest for this loan."
MsgBox Msg                                ' Display results.
```

IRR Function

Description Returns the internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax **IRR**(*valuearray*( ), *guess*)

Remarks The internal rate of return is the interest rate received for an investment consisting of payments and receipts that occur at regular intervals.

The **IRR** function uses the following arguments.

Argument	Description
valuearray( )	Array containing cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).
guess	Value you guess will be returned by **IRR.** In most cases, *guess* is 0.1 (10 percent).

The **IRR** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence. The cash flow for each period doesn't have to be constant, as it would for an annuity.

IRR is calculated by iteration. Starting with the value of *guess*, **IRR** cycles through the calculation until the result is accurate to within 0.00001 percent. If, after 20 tries, it can't find a result, **IRR** fails.

Because the **IRR** function requires an array of cash flows, it cannot be used as an expression on a form. It can be used only in Access Basic.

See Also

MIRR Function, **NPV** Function, **Rate** Function

Example

This example returns the internal rate of return for a series of five cash flows contained in the array `Values()`. The first array element is a negative cash flow representing business start-up costs. The remaining four cash flows represent positive cash flows for the subsequent four years. `Guess` is the estimated internal rate of return.

```
Static Values(5) As Double                  ' Set up array.
Guess = .1                                  ' Guess starts at 10%.
Fmt = "#0.00"                               ' Define percentage format.
Values(0) = -70000                          ' Business start-up costs.
Values(1) = 22000 : Values(2) = 25000       ' Positive cash flows reflecting
Values(3) = 28000 : Values(4) = 31000       '  income for four successive years.
RetRate = IRR(Values(), Guess) * 100        ' Calculate internal rate.
Msg = "The internal rate of return for these five cash flows is "
Msg = Msg & Format(RetRate, Fmt) & " percent."
MsgBox Msg                                  ' Display internal return rate.
```

Is Operator

Description Used with the reserved word **Null** to determine whether an expression is **Null**.

Syntax *expr* **Is [Not] Null**

Remarks The **Is** operator appears only with the **Null** reserved word. The following example shows how you can evaluate a control. If the control is **Null**, the **IIf** function returns `"No Entry"`; otherwise, it returns the control's value.

```
=IIf(Forms![My Form]![My Control] Is Null, "No Entry",
➥ Forms![My Form]![My Control])
```

You can include the **Not** logical operator to evaluate the opposite condition (that is, whether an expression is not **Null**).

You can use the **Is** operator in a query expression or in a calculated control on a form or report.

See Also **IsNull** Function, **Not** Operator

Example The following example uses the Orders table in the NWIND.MDB database to create a query that includes orders for which the `Ship Region` is not **Null**. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT * FROM Orders WHERE [Ship Region] Is Not Null;
```

The next example sets the text for a control on a form based on the Orders table in the NWIND.MDB database. Depending on whether the region is present, it sets the text for the control. You can enter this expression in a calculated control on the form.

```
=IIf([Ship Region] Is Null, "Ship Region is missing", "Ship Region
➥ is present")
```

IsDate Function

Description Returns a value indicating whether or not a **Variant** argument can be converted to a date.

Syntax **IsDate**(*variant*)

Remarks The argument *variant* can be any **Variant** expression of **VarType** 7 (Date) or of **VarType** 8 (**String**). The **IsDate** function returns **True** if the **Variant** can legally be converted to a date; otherwise, it returns **False**. The range of valid dates is January 1, 100 A.D. through December 31, 9999 A.D.

See Also **CVDate** Function, **IsEmpty** Function, **IsNull** Function, **IsNumeric** Function, **VarType** Function

Example This example evaluates `TestVar` to determine whether it can legally be converted to a numeric data type and displays an appropriate message.

```
TestVar = InputBox("Enter a string you'd like to display as a date.")
If IsDate(TestVar) Then                    ' Test variable.
  MsgBox "The date is: " & Format(CVDate(TestVar),"dddddd")
Else
  MsgBox "TestVar doesn't make sense as a date."
End If
```

IsEmpty Function

Description Returns a value indicating whether or not a **Variant** variable has been initialized.

Syntax **IsEmpty**(*variant*)

Remarks The argument *variant* can be any **Variant** expression. However, because **IsEmpty** is used to determine whether or not variables are initialized, the argument is most often a single variable name. The **IsEmpty** function returns **True** if the **Variant** contains the Empty value; otherwise, it returns **False**. When an expression contains more than one **Variant**, **IsEmpty** returns **True** only when all of the variables in the expression are Empty.

The Empty value indicates that a **Variant** has not been initialized. Empty is not the same as **Null**, which indicates that a **Variant** contains no data.

See Also **IsDate** Function, **IsNull** Function, **IsNumeric** Function, **VarType** Function

Example This example uses **IsEmpty** to determine whether `TestVar` has been initialized. If it hasn't been initialized, the user is asked whether it should be.

```
Dim TestVar As Variant                                    ' Declare a variable.
If IsEmpty(TestVar) Then                                  ' Test variable.
  InitVar = MsgBox("TestVar is uninitialized. Initialize now?", 36)
  If InitVar = 6 Then                                     ' If Yes,
    TestVar = 1.25                                        ' initialize TestVar.
    MsgBox "TestVar initialized to " & TestVar            ' Display confirmation.
  End If
End If
```

IsNull Function

Description Returns a value that indicates whether or not a **Variant** contains the special **Null** value.

Syntax **IsNull**(*variant*)

Remarks The argument *variant* can be any **Variant** expression. The **IsNull** function returns **True** if the expression contains the **Null** value; otherwise, it returns **False**.

The **Null** value indicates that the **Variant** contains no data. **Null** is not the same as Empty, which indicates that a **Variant** has not yet been initialized. It is also not the same as a zero-length string, which is often referred to as a null string.

Caution Using the **IsNull** function is the only way from within Access Basic to determine whether or not a **Variant** expression contains a **Null** value. Expressions that you might expect to evaluate **True** under some circumstances, such as `If var = Null` and `If var <> Null`, are always **False**. Because of **Null** propagation, any expression containing a **Null** is itself **Null** and therefore **False**.

See Also **IsDate** Function, **IsEmpty** Function, **IsNumeric** Function, **VarType** Function

Example This example evaluates `TestVar` to determine whether it contains the **Null** value and then displays an appropriate message. The variable `TestVar` starts as **Null** but is changed to a zero-length string the first time through the loop. The second time through the loop, `TestVar` again becomes **Null** and the loop exits.

```
Dim TestVar As Variant                  ' Declare variable.
TestVar = Null                          ' Initialize as Null.
Do
  If IsNull(TestVar) Then               ' Evaluate variable.
    MsgBox "TestVar is Null."           ' Indicate if Null.
    TestVar = ""                        ' Make zero-length string.
  Else
    MsgBox "TestVar is not Null."       ' Indicate if not Null.
    TestVar = Null                      ' Make Null.
  End If
Loop Until IsNull(TestVar)              ' Loop until TestVar is Null.
```

You can use **IsNull** to determine whether a control or field contains data. This example could be used in a macro condition to validate the contents of the `First Name` control on the `Employees` form. If the expression evaluates true (nonzero), a message could be displayed to alert the user that the field contains no data.

```
IsNull(Forms!Employees![First Name])
```

IsNumeric Function

Description Returns a value indicating whether or not a **Variant** variable can be converted to a numeric data type.

Syntax **IsNumeric**(*variant*)

Remarks The argument *variant* can be any **Variant** expression. The **IsNumeric** function returns **True** if the expression can legally be converted to a number; otherwise, it returns **False**. Expressions that can legally be converted to a numeric data type include **Variant** variables of **VarType** 0 (Empty), of any numeric value (**VarType** 2–6), of **VarType** 7 (Date), and of **VarType** 8 (**String**) if the **String** can be interpreted as numeric.

See Also **IsDate** Function, **IsEmpty** Function, **IsNull** Function, **VarType** Function

Example This example evaluates `TestVar` to determine whether it can legally be converted to a numeric data type and displays an appropriate message.

```
TestVar = InputBox$("Please enter a number, letter, or symbol.")
If IsNumeric(TestVar) Then                    ' Evaluate variable.
  MsgBox "TestVar data is numeric."           ' Message if number.
Else
  MsgBox "TestVar data is not numeric."       ' Message if not.
End If
```

Item Property

Applies To Control (unbound object frame (linked)).

Description Contains a description of the data displayed in a linked object frame.

Setting The Item property setting is a **String** value.

Remarks The description of a linked object frame depends on the kind of OLE object it is linked to. For example, if you link it to cells in a Microsoft Excel worksheet, Microsoft Access displays the cells in the object frame, and the Item property identifies the linked cells using the row-and-column format. For a chart, the Item property setting is the word Chart. For a picture, it contains the coordinates of the linked picture area.

Note The data for a linked OLE object is displayed in the frame but is stored in the application that creates the data. To change the data, you use that application.

When you link an OLE object to an object frame using the Paste Link command on the Edit menu, Microsoft Access automatically sets the SourceObject, OLEClass, and Item properties. The following table shows typical property settings for various OLE objects.

Object	SourceObject property	OLEClass property	Item property
Worksheet	C:\EXCEL\XLS\MYWRKSHT.XLS	Microsoft Excel Worksheet	R1C1:R7C4
Chart	C:\EXCEL\XLS\MYCHART.XLC	Microsoft Excel Chart	Chart
Picture	C:\WINDOWS\MYBITMAP.BMP	Paintbrush Picture	0 0 72 71

Linking an OLE object is not the same as embedding it.

Access Basic

The property setting is a **String** value.

Design view: read-only. Other views: read-only.

KeepTogether Property

Applies To

Form and report sections (all except page headers and page footers).

Description

Determines whether Microsoft Access prints an entire section on one page.

Setting

The KeepTogether property settings are:

Setting	Description
Yes	Microsoft Access begins printing the section at the top of the next page if it can't print all of it on the current page.
No	(Default) Microsoft Access prints as much of the section as possible on the current page and continues printing it on the next page.

Remarks

If a section is longer than one page, Microsoft Access continues printing it on the next page regardless of the KeepTogether setting.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also CanGrow, CanShrink Properties; ForceNewPage Property; OnFormat Property

Kill Statement

Description Deletes files from a disk.

Syntax **Kill** *filespec*

Remarks The **Kill** statement is similar to the operating system commands **erase** and **del**.

You can use **Kill** for all types of disk files: program files, random-access data files, and sequential data files. The argument *filespec* is a string expression that specifies a path or file name. The path and file name can include a drive specification and any valid wildcard characters.

Kill deletes files only. To delete directories, use the operating system command **rmdir** or the Access Basic **RmDir** statement.

Using **Kill** to delete a file that is currently open by Access Basic produces an error.

Warning Be extremely careful when using wildcards with **Kill**. You can delete more files than you intended.

See Also **RmDir** Statement

Example This example uses **Kill** to remove a user-specified file.

```
On Error GoTo Errhandler                    ' Set up error handler.
DelFile = UCase(InputBox("Enter file name you want deleted."))
If Len(DelFile) Then                         ' Check for entry.
  Ansr = MsgBox("Sure you want to delete " & DelFile & "?", 4)
  If Ansr = 6 Then                           ' User chose "Yes."
    Msg = "Deleting " & DelFile & " from your disk."
    Kill DelFile                             ' Delete file from disk.
  Else
    Msg = DelFile & " was not deleted."
  End If
```

```
Else
  Msg = "You didn't enter a file name."
End If
MsgBox Msg                                    ' Display message.
End

Errhandler:
  If Err = 53 Then                            ' Error 53 is "File not Found".
    Msg = "Sorry, the file you named could not be found."
  Else
    Msg = "Sorry, unable to delete file."
  End If
  Resume Next
```

LabelAlign, TextAlign Properties

Apply To

- LabelAlign—tools (bound object frame, check box, combo box, command button, graph, list box, option button, option group, subform/subreport, text box, toggle button).
- TextAlign—tools and controls (combo box, label, text box).

Description

- LabelAlign—specifies the text alignment in attached labels for new controls; affects the implementation of LabelX.
- TextAlign—for toolbox items, specifies the text alignment in new controls; for controls, specifies the text alignment.

Setting

To see the property sheet, choose the Properties button on the tool bar or the Properties command from the View menu. Select the tool or control.

The LabelAlign and TextAlign property settings are:

Setting	Description
General	(Default) Text aligns to the left, and numbers and dates align to the right. Label text aligns to the left.
Left	Text aligns to the left.
Center	Text is centered.
Right	Text aligns to the right.

Remarks

When created, most controls have an attached label. Changes to this default control property setting affect only controls created on the current form or report. To change the default control property settings for all new forms or reports, you must create a new template.

Access Basic

The LabelAlign property isn't available from Access Basic.

The TextAlign property can't be used to set a default control property in the toolbox. For controls, the TextAlign property settings and their values are:

Setting	Value
General	0
Left	1
Center	2
Right	3

Design view: read/write. Other views: read-only.

See Also

AddColon, AutoLabel Properties; LabelX, LabelY Properties

LabelX, LabelY Properties

Apply To

Tools (bound object frame, check box, combo box, command button, graph, list box, option button, option group, subform/subreport, text box, toggle button).

Description

Specify the placement of the label for a new control.

Setting

Use the LabelX and LabelY properties to change the placement of labels attached to new controls.

Enter a positive or negative number that specifies the starting point of label text relative to the upper-left corner of the label's attached control. To use a unit of measurement different from the setting in the Microsoft Windows Control Panel, specify the unit (for example, 1.5 in or 2 cm).

To see the property sheet, choose the Properties button on the tool bar or the Properties command from the View menu. Select the tool in the toolbox.

Remarks

LabelX represents the horizontal axis and LabelY represents the vertical axis. A negative number for LabelX places the label to the left of the control. A negative number for LabelY places the label above the control. Positive numbers place the label to the right and below the control.

If the LabelAlign property is set to Right, the text for LabelX starts at the right of the attached label. If LabelAlign is set to Center, the text starts at the center of the attached label. Combining LabelX, LabelY, and LabelAlign properties produces the following results.

LabelX	LabelY	LabelAlign	Result
–1 in	0 in	General	Label's upper-left corner is 1 inch to the left of the control's upper-left corner.
–.5 in	0 in	Right	Label's upper-right corner is 0.5 inch to the left of the control's upper-left corner.
.5 in	–.5 in	Center	Center of the top of the label is 0.5 inch to the right and 0.5 inch above the control's upper-left corner.

Changes to default control properties are in effect only for the current form or report. To change default values for all new forms or reports, you must create a new template.

Access Basic These properties aren't available from Access Basic.

See Also AddColon, AutoLabel Properties; LabelAlign, TextAlign Properties

Last Function

See **First**, **Last** Functions

LastUpdated Property

Applies To **Tables**.

Description Contains the date and time of the most recent change to a table in Design view. This property can be used only in Access Basic code.

Setting The LastUpdated property setting is a **Variant** of **VarType** 7 (Date).

Usage Read-only.

Remarks In a multiuser environment, the date and time are taken from the computer on which the table design was updated. If the various users maintain different dates and times on their workstations, the LastUpdated property setting could be incorrect. When possible, each user should get the date and time directly from the file server.

See Also DateCreated Property

LayoutForPrint Property

Applies To Forms. Reports.

Description Determines whether the form or report uses printer or screen fonts.

Setting The LayoutForPrint property settings are:

Setting	Description
Yes	(Default for reports) Use printer fonts.
No	(Default for forms) Use screen fonts.

Remarks When you choose a font for a form or a report, the characters on the screen may not look exactly like those that appear on the printed page because printer fonts and screen fonts can differ.

Tip If you select a scalable font, such as a TrueType font, the screen and printer characters will look the same.

Screen fonts are the letters, numbers, and symbols displayed on the screen. The screen fonts Microsoft Access uses are those that were installed when you installed the Microsoft Windows graphical environment. If you set up a printer, additional screen fonts may have been installed automatically. You may also have added additional screen fonts on your own.

Printer fonts are the letters, numbers, and symbols that are printed when you print a report or a form. The fonts used depend on your printer.

If you set the LayoutForPrint setting to Yes, the tool bar displays the fonts and point sizes available for the printer you selected using the Print Setup command on the File menu.

After you select a printer, design a form or report, and then save the form or report, Microsoft Access remembers the type of printer that should be used for that form or report. If you print the form or report on a different type of printer, Microsoft Access displays a message to let you know that the form or report was designed for another kind of printer. If you print the form or report anyway, your printer may substitute fonts if it doesn't have all the fonts specified in the form or report design.

If you change the LayoutForPrint setting after you design the form or report, you may find that some of the fonts aren't available. For example, certain screen fonts may not be available for the selected printer. You can reselect the font for each control, making sure it is still available. If you don't check the fonts, Microsoft Access may have to make substitutions.

See Also FontName, FontSize Properties

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

LBound Function

Description Returns the smallest available subscript for the indicated dimension of an array.

Syntax **LBound**(*array* [, *dimension*])

Remarks This function can be used only in Access Basic code.

The **LBound** function is used with the **UBound** function to determine the size of an array. Use the **UBound** function to find the upper limit of an array dimension.

LBound uses the following arguments.

Argument	Description
array	Name of an array variable.
dimension	Integer indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second dimension, and so on. If *dimension* is omitted, 1 is assumed.

LBound returns the values listed in the table below for an array with the following dimensions:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

Statement	Value returned
LBound(A, 1)	1
LBound(A, 2)	0
LBound(A, 3)	–3

The default lower bound for any dimension is either 0 or 1, depending on the setting of the **Option Base** statement.

Arrays for which dimensions are set using the **To** clause in a **Dim**, **Global**, **ReDim**, or **Static** statement can have any integer value as a lower bound.

See Also **Dim** Statement, **Global** Statement, **Option Base** Statement, **ReDim** Statement, **Static** Statement, **UBound** Function

Example In this example, the **LBound** function determines the lower bound for a two-dimensional array.

```
A = Int(9 * Rnd + 2)                              ' First dimension.
B = Int(9 * Rnd + 2)                              ' Second dimension.
ReDim Array(A To 20, B To 20) As Integer          ' Set dimensions.
MsgBox "Dimension 1 = " & LBound(Array, 1)        ' First LBound.
MsgBox "Dimension 2 = " & LBound(Array, 2)        ' Second LBound.
```

LCase, LCase$ Functions

Description Return a string in which all letters of an argument have been converted to lowercase.

Syntax **LCase**[$](*strexpr*)

Remarks **LCase** returns a **Variant**; **LCase$** returns a **String**.

The argument *strexpr* can be any string expression. However, only **LCase** can accept a **Variant** of **VarType** 1 (**Null**) as *strexpr*, in which case, a **Null** is returned.

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

See Also **UCase**, **UCase$** Functions

Example This example returns an all-lowercase version of the argument string.

```
= LCase("ONCE UPON A TIME")
```

Left, Left$ Functions

Description Return the leftmost *n* characters of a string argument.

Syntax **Left**[**$**](*strexpr*, *n*)

Remarks **Left** returns a **Variant**; **Left$** returns a **String**.

The argument *strexpr* can be any string expression. However, only **Left** can accept a **Variant** of **VarType** 1 (**Null**) as *strexpr*, in which case, a **Null** is returned.

The argument *n* is a **Long** expression indicating how many characters to return. It must be between 0 and approximately 65,535, inclusive. If *n* is 0, the return value is a zero-length string. If *n* is greater than or equal to the number of characters in *strexpr*, the entire string is returned.

To find the number of characters in *strexpr*, use **Len**(*strexpr*).

See Also **Len** Function; **Mid**, **Mid$** Functions; **Right**, **Right$** Functions

Example This example returns the two leftmost characters of a variable called `MyText`.

```
= Left(MyText, 2)
```

Left, Top Properties

Apply To Controls (all).

Description

- Left—specifies the position of an object on the horizontal axis.
- Top—specifies the position of an object on the vertical axis.

Setting In the property sheet, enter a number for the desired position in the current measure. To use a unit of measurement other than the default, include an indicator, such as "cm" or "in."

Remarks A control's location is measured from the center of its left or top border to the section's left or top border. When you move a control, its new Left and Top coordinates are automatically entered in the property sheet.

The left margin on a form or report is determined by the Left property setting of the leftmost section on the form or report. For example, if a report has three sections with Left property settings of 1, 1, and 2, the left margin will be 1.

The top margin on a form or report is determined by the Top property setting of the first section on the form or report. For example, if a form has only a detail section, the form's top margin will be the same as that section's Top property setting.

Note To set the right and bottom margins, use the Height and Width properties.

Sections can use the Top and Left properties only while printing. You must use a macro or Access Basic to specify the position of the section on the printed page. For forms and reports, these properties are available only from a macro or Access Basic and are read-only. However, you can reposition forms and reports using the MoveSize action.

Access Basic

Use a numeric expression to set the value of this property.

Design view: read/write. Other views: read-only.

Note Values are expressed in twips (1 twip equals 1/1440 of an inch, or about 1/567 of a centimeter).

See Also Height, Width Properties

Len Function

Description Returns the number of characters in a string expression or the number of bytes required to store a variable.

Syntax 1 **Len**(*strexpr*)

Syntax 2 **Len**(*variablename*)

Remarks If you use the first syntax, **Len** returns the number of characters in the argument *strexpr*.

If you use the second syntax, **Len** returns the number of bytes required to store a variable of the given data type. If *variablename* is a **Variant**, **Len** always returns the number of bytes required to store the **Variant** data as a **String**, regardless of the **VarType**.

Because **Len** works with user-defined data types as well as fundamental Access Basic data types, the second syntax is particularly useful for determining record size when you are performing file input/output with random-access files. However, if a user-defined data type contains **Variant** or variable-length **String** elements, **Len** may not be able to properly determine the actual number of storage bytes required.

If the argument to **Len** is **Null**, the function returns a **Null**.

See Also **InStr** Function

Example This example shows how **Len** can be used to return the number of bytes needed to store each fundamental Access Basic data type.

```
= Len(A)                    ' Returns length of Variant converted to String.
= Len(B%)                   ' Returns length of Integer.
= Len(C&)                   ' Returns length of Long.
= Len(D!)                   ' Returns length of Single.
= Len(E#)                   ' Returns length of Double.
= Len(F@)                   ' Returns length of Currency variable.
= Len("This text")          ' Returns the length of String.
```

Let Statement

Description Assigns the value of an expression to a variable or assigns an object reference to a variable.

Syntax **[Let]** *variable* = *valueexpression*

Remarks The **Let** statement uses these arguments.

Argument	Description
variable	Name of the variable
=	Value assignment operator
valueexpression	Value assigned to the variable

The reserved word **Let** is always optional. You can assign a **String** or numeric expression to a variable without using the **Let** reserved word. In fact, most Basic programmers never use the **Let** reserved word.

A value expression can be assigned to a variable only if the data types they have are compatible. You can't assign **String** expressions to numeric variables, and you can't assign numeric expressions to **String** variables. If you do, a `Type mismatch` error occurs.

Variant variables can be assigned either **String** or numeric expressions. However, the reverse is not true. Any **Variant** except a **Null** can be assigned to a **String** variable, but only a **Variant** that contains a value that can be interpreted as a number can be assigned to a numeric variable. Use the **IsNumeric** function to determine if the **Variant** can be converted to a number.

Caution Assigning a value expression of one numeric data type to a variable of a different numeric data type coerces the value of the expression into the data type of the resulting variable.

Let statements can be used to assign one record variable to another only when both variables are of the same user-defined type. Use the **LSet** statement to assign record variables of different user-defined types.

See Also **IsNumeric** Function, **LSet** Statement

Example This example uses statements with and without the **Let** reserved word to assign the value of an expression to a variable.

```
Let Pi = 4 * Atn(1)
Pi = 3.1415
```

Like Operator

Description Used to compare two string expressions.

Syntax *result* = *expression* **Like** *pattern*

Remarks If *expression* matches *pattern*, *result* is **True** (–1); if there is no match, *result* is **False** (0); and if either *expression* or *pattern* is a **Null**, *result* is also a **Null**. The case sensitivity and character sort order of the **Like** operator depend on the setting of the **Option Compare** statement. However, if the module doesn't contain an **Option Compare** statement, the default string-comparison method is **Binary**.

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to use wildcard characters, like those recognized by the operating system, to match strings. The following table shows the wildcard characters you can use and the number of digits or strings they match.

Character(s) in *pattern*	Matches in *expression*
?	Any single character
*	Zero or more characters
#	Any single digit (0–9)
[*charlist*]	Any single character in *charlist*
[!*charlist*]	Any single character not in *charlist*

A group of one or more characters (*charlist*) enclosed in brackets ([ ]) can be used to match any single character in *expression* and can include almost any characters in the ANSI character set, including digits. In fact, the special characters left bracket ([ ),

question mark (?), number sign (#), and asterisk (*) can be used to match themselves directly only by enclosing them in brackets. The right bracket ( ]) cannot be used within a group to match itself, but it can be used outside a group as an individual character.

In addition to a simple list of characters enclosed in brackets, *charlist* can specify a range of characters by using a hyphen (-) to separate the upper and lower bounds of the range. For example, `[A-Z]` in *pattern* results in a match if the corresponding character position in *expression* contains any of the uppercase letters in the range A through Z. Multiple ranges are included within the brackets without any delimiting. For example, `[a-zA-Z0-9]` matches any alphanumeric character.

Other important rules for pattern matching include the following:

- An exclamation mark (!) at the beginning of *charlist* means that a match is made if any character except the ones in *charlist* are found in *expression*. When used outside brackets, the exclamation mark matches itself.
- The hyphen (-) can appear either at the beginning (after an exclamation mark if one is used) or at the end of *charlist* to match itself. In any other location, the hyphen is used to identify a range of ANSI characters.
- When a range of characters is specified, they must appear in ascending sort order (from lowest to highest). `[A-Z]` is a valid pattern, but `[Z-A]` is not.
- The character sequence `[ ]` is ignored; it is considered to be a zero-length string.

See Also Operator Precedence, **Option Compare** Statement

Example The following table shows how you can use **Like** to test expressions for different patterns.

Kind of matching	With this pattern	This expression returns True (–1)	This expression returns False (0)
Multiple characters	`"a*a"`	`"aa", "aBa", "aBBBa"`	`"aBC"`
Special character	`"a[*]a"`	`"a*a"`	`"aaa"`
Multiple characters	`"ab*"`	`"abcdefg", "abc"`	`"cab", "aab"`
Single character	`"a?a"`	`"aaa", "a3a", "aBa"`	`"aBBBa"`
Single digit	`"a#a"`	`"a0a", "a1a", "a2a"`	`"aaa", "a10a"`
Range of characters	`"[a-z]"`	`"f", "p", "j"`	`"2", "&"`
Outside a range	`"[!a-z]"`	`"9", "&", "%"`	`"b", "a"`
Not a digit	`"[!0-9]"`	`"A", "a", "&", "~"`	`"0", "1", "9"`
Combined	`"a[!b-m]#"`	`"An9", "az0", "a99"`	`"abc", "aj0"`

LimitToList Property

Applies To Control (combo box).

Description Determines whether a combo box accepts any entered text or only text that matches one of the listed choices.

Setting The LimitToList property settings are:

Setting	Description
Yes	If the user selects an item from the list or enters text that matches a listed item, Microsoft Access accepts it. Entered text that doesn't match a listed item isn't accepted. The user must retype the entry, select a listed item, press ESC, or choose the Undo Current Field/Current Record command from the Edit menu.
No	(Default) Microsoft Access accepts any text that conforms to the ValidationRule.

Remarks You can use this property to limit combo box values to the listed items. If you set the property to No, newly entered text is not permanently added to the combo box list. To add an item to the list, use a macro or a procedure.

Note If you bind a combo box column to a hidden column (column width = 0), Microsoft Access automatically sets the LimitToList property of the combo box to Yes.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

Line Input # Statement

Description Reads a line from a sequential file into a **String** or **Variant** variable.

Syntax **Line Input #** *filenumber, variablename*

Remarks The **Line Input #** statement uses these arguments.

Argument	Description
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file. Note that the number sign (#) preceding *filenumber* is not optional.
variablename	Name of the variable used to receive a line of text from the file.

The **Line Input #** statement reads all characters in a sequential file up to a carriage return. It then skips over the carriage-return/linefeed sequence.

Line Input # is used to read a text file one line at a time.

See Also **Input #** Statement

Example In this example, the **Line Input #** statement reads the CONFIG.SYS file and displays the first line.

```
Open "C:\CONFIG.SYS" For Input As #1        ' Open file.
Line Input #1, TextLine                     ' Get complete line.
Close #1                                    ' Close file.
MsgBox TextLine                             ' Display message.
```

Line Method

Description Draws lines and rectangles on a **Reports** object.

Syntax *object*.**Line** [[**Step**](*x1*, *y1*)] – [**Step**](*x2*, *y2*) [,[*color*][,**B**[**F**]]]

Remarks You can use this method only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat or OnPrint.

The **Line** method uses these arguments.

Argument	Description
object	**Reports** object on which the line or rectangle is to be drawn.
Step	Reserved word that specifies that the starting point coordinates are relative to the current graphics position given by the CurrentX and CurrentY property settings of *object*.
x1, *y1*	**Single** values indicating the coordinates of the starting point for the line or rectangle. The scale properties (ScaleMode, ScaleLeft, ScaleTop, ScaleHeight, and ScaleWidth) of *object* determine the unit of measure used. If this argument is omitted, the line begins at the position indicated by CurrentX and CurrentY.
Step	Reserved word that specifies that the end-point coordinates are relative to the line's starting point.
x2, *y2*	**Single** values indicating the coordinates of the end point for the line to draw. These coordinates are required.
color	**Long** value indicating the RGB color used to draw the line. If this argument is omitted, the ForeColor property is used. You can use the **RGB** function or **QBColor** function to specify the color.
B	Option that causes a box to be drawn using the coordinates that specify opposite corners of the box.
F	**F** cannot be used without **B**. If the **B** option is used, the **F** option specifies that the box is filled with the same color used to draw the box. If **B** is used without **F**, the box is filled with the color specified by the current FillColor and BackStyle property settings. The default value for BackStyle is Normal.

When drawing lines that are connected, make sure that each noninitial line begins at the end point of the previous line.

The width of the line drawn depends on the DrawWidth property setting. The way a line or box is drawn on the background depends on the settings of the DrawMode and DrawStyle properties.

When you apply the **Line** method, CurrentX and CurrentY are set to the end point specified by the arguments.

See Also BackStyle Property; CurrentX, CurrentY Properties; DrawMode Property; DrawStyle Property; DrawWidth Property; FillColor Property; ForeColor Property; **QBColor** Function; **RGB** Function; ScaleHeight, ScaleWidth Properties; ScaleLeft, ScaleTop Properties; ScaleMode Property

Example This example uses the **Line** method to draw a red box five pixels inside the edge of a report named Report1. The **RGB** function is used to make the line red.

```
Dim Rpt As Report                          ' Dimension variable.
Set Rpt = Reports!Report1                  ' Assign variable to Report.
Rpt.ScaleMode = 3                          ' Set scale to pixels.
InTop = Rpt.ScaleTop + 5                   ' Top inside edge.
InLeft = Rpt.ScaleLeft + 5                 ' Left inside edge.
InWidth = Rpt.ScaleWidth - 10              ' Width inside edge.
InHeight = Rpt.ScaleHeight - 10            ' Height inside edge.
Color = RGB(255,0,0)                       ' Make color red.
' Draw line as a box.
Rpt.Line (InTop, InLeft) - (InWidth, InHeight), Color, B
```

LineSlant Property

Applies To Control (line).

Description Specifies whether a line slants from upper left to lower right or from upper right to lower left.

Setting The LineSlant property settings are:

Setting	Description
\	Upper left to lower right
/	Upper right to lower left

Remarks Use the LineSlant property to quickly change a line's direction. To position and size the line on your form or report, use the mouse.

Access Basic

The property settings and their values are:

Setting	Value
\	0
/	–1

Design view: read/write. Other views: read-only.

LinkChildFields, LinkMasterFields Properties

Apply To

Controls (graph, subform/subreport, unbound object frame [embedded]).

Description

These properties together define how Microsoft Access matches records in a form or report to records in an embedded object, such as a graph. Each property identifies one or more fields in one object that matches one or more fields in the other object.

- LinkChildFields—identifies fields in the child, or embedded, object.
- LinkMasterFields—identifies fields in the master form or report.

Setting

In the property sheets for the child object and the master object, enter one of the following:

- The name of a field contained in the object's underlying table or query (identified by its RecordSource property).
- A list of fields separated by semicolons.

The number of fields and the data types must match in both property settings.

Remarks

When you drag a form from the Database window onto another form to create a subform, Microsoft Access automatically tries to establish a link between the forms. This is also true when you drag a form or report onto a report. Microsoft Access establishes a link if:

- Both the main form and subform are based on tables and a relationship has been defined with the Relationships command.
- The main form and subform contain fields with the same name and data type, and the field on the main form is the primary key of the underlying table.

If Microsoft Access finds a relationship or a match, these properties automatically show the field names that define the link. You should verify the validity of an automatic link. If the main form is based on a query, or if neither of the preceding conditions is true, Microsoft Access cannot automatically match fields to create a link.

The LinkChildFields and LinkMasterFields property settings must have the same number of fields and must represent data of the same type. For example, if a Products table and a Suppliers table each have a Supplier field that contains the same type of data, you would

enter Supplier for both properties. The subform could automatically display all the products available from the supplier identified in the Supplier field on the main form.

Although the data must match, the names of the fields can differ. For example, the Supplier ID field from the Suppliers table could be linked to the Suppliers field from the Products table.

Note A relationship between underlying tables or queries can be either a one-to-many relationship, in which one record in the master table relates to one or more records in the child table, or a one-to-one relationship, in which one record in the master table matches only one record in the child table.

Access Basic

Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

LinkMasterFields Property

See LinkChildFields, LinkMasterFields Properties

ListFields Method

Description Creates a **Snapshot** with one record for each field in a specified recordset.

Syntax **Set** *snapshot* = *recordset*.**ListFields( )**

Remarks The **ListFields** method uses the following arguments.

Argument	Description
snapshot	Name of the **Snapshot** you want to create
recordset	Name of an existing recordset that contains the fields you want to list

ListFields can return up to 255 records. The records in the **Snapshot** are in the same order as the fields in *recordset*. The following table shows the **Snapshot** field contents.

Field	Access Basic data type	Description
Name	**String**	Name of the field
Type	**Long**	Field data type: 1 Yes/No 2 Number (Byte) 3 Number (Integer) 4 Number (Long) 5 Currency 6 Number (Single) 7 Number (Double) 8 Date/Time 9 Reserved for future use 10 Text 11 OLE Object 12 Memo
Size	**Long**	Maximum size of the field (see note)
Attributes	**Long**	Bit field: 1 Fixed 16 Auto Increment 32 Updatable
SourceTable	**String**	Name of the underlying table the field came from
SourceField	**String**	Name of the field in the underlying table
CollatingOrder	**Integer**	Comparison code

Notes

- When you use **ListFields** to create a **Snapshot** that contains a list of field names, you can inspect the Type field using the Type constants. To inspect the Attributes field, you can use the Attributes constants.
- For Memo and OLE Object fields, Size contains 0; for bit fields, it contains 1.
- You can't use **ListFields** or any of the Find methods on a **Snapshot** created with **ListFields, ListParameters, ListIndexes,** or **ListTables**.

See Also **ListIndexes** Method, **ListParameters** Method, **ListTables** Method

Example This example creates a **Snapshot** that lists all of the fields in the Employees table of the NWIND.MDB database.

```
Dim ListSet As Snapshot, MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Employees")       ' Open Table.
Set ListSet = MyTable.ListFields()              ' Copy field info to ListSet.
MyTable.Close                                   ' Close Table.
```

`ListSet` might look like the following table.

Name	Type	Size	Attributes	SourceTable	SourceField
Employee ID	4	4	49	Employees	Employee ID
Last Name	10	20	32	Employees	Last Name
First Name	10	10	32	Employees	First Name
Title	10	30	32	Employees	Title
Birth Date	8	8	33	Employees	Birth Date
Hire Date	8	8	33	Employees	Hire Date
Address	10	60	32	Employees	Address
City	10	15	32	Employees	City
Region	10	15	32	Employees	Region
Postal Code	10	10	32	Employees	Postal Code
Country	10	15	32	Employees	Country
Home Phone	10	24	32	Employees	Home Phone
Extension	10	4	32	Employees	Extension
Photo	11	0	32	Employees	Photo
Notes	12	0	32	Employees	Notes

ListIndexes Method

Description Creates a **Snapshot** with one record for each index in a specified table.

Syntax **Set** *snapshot* = *table*.**ListIndexes( )**

Remarks The **ListIndexes** method uses the following arguments.

Argument	Description
snapshot	Name of the **Snapshot** you want to create
table	Name of an existing table object that contains the indexes you want to list

The snapshot records are sorted by index name and (for indexes with multiple-field keys) field order of keys in the index. The following table shows the **Snapshot** field contents.

Field	Access Basic data type	Description
IndexName	**String**	Name of the index
FieldCount	**Long**	Number of fields in the index
IndexAttributes	**Long**	Bit field: 1 Unique 2 Primary 4 DisallowNull 8 IgnoreNull
FieldName	**String**	Name of the field in the table
FieldOrder	**Long**	Order of the field in the table
FieldType	**Long**	Field data type: 1 Yes/No 2 Number (Byte) 3 Number (Integer) 4 Number (Long) 5 Currency 6 Number (Single) 7 Number (Double) 8 Date/Time 9 Reserved for future use 10 Text 11 OLE Object 12 Memo
FieldAttributes	**Long**	Bit field: 1 descending
FieldCollatingOrder	**Integer**	Comparison code

Note When you use **ListIndexes** to create a **Snapshot** that contains a list of indexes, you can inspect the IndexAttributes field using the IndexAttributes constants. To inspect the FieldAttributes field, you can use the FieldAttributes constant.

You can't use the Find methods on a **Snapshot** created by **ListIndexes**.

If the specified table has no indexes, the **Snapshot** will have no records.

See Also **ListFields** Method, **ListParameters** Method, **ListTables** Method

Example This example creates a **Snapshot** that lists all of the indexes in the Customers table of the NWIND.MDB database.

```
Dim ListSet As Snapshot, MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Customers")     ' Open Table.
Set ListSet = MyTable.ListIndexes()           ' Copy index info to ListSet.
MyTable.Close                                 ' Close Table.
```

`ListSet` might look like the following table.

IndexName	FieldCount	IndexAttributes	FieldName	FieldOrder	FieldType	FieldAttributes
City	1	4	City	6	10	0
Company Name	1	4	CompanyName	2	10	0
PrimaryKey	1	3	Customer ID	1	10	0

ListParameters Method

Description Creates a **Snapshot** with one record for each parameter in a specified **QueryDef** object.

Syntax **Set** *snapshot* = *querydef*.**ListParameters( )**

Remarks The **ListParameters** method uses the following arguments.

Argument	Description
snapshot	Name of the **Snapshot** you want to create
querydef	Name of an open **QueryDef** object that contains the parameters you want to list

The records in the **Snapshot** are in the same order as the parameters in the specified **QueryDef**. The following table shows the **Snapshot** field contents.

Item	Access Basic data type	Description
Name	**String**	Name of the parameter
Type	**Long**	Field data type: 1 Yes/No 2 Number (Byte) 3 Number (Integer) 4 Number (Long) 5 Currency 6 Number (Single) 7 Number (Double) 8 Date/Time 9 Reserved for future use 10 Text 11 OLE Object 12 Memo

You can't use the Find methods on a **Snapshot** created by **ListParameters**.

If the specified **QueryDef** contains no parameters, the **Snapshot** will have no records.

See Also **ListFields** Method, **ListIndexes** Method

Example Using the Orders table in the NWIND.MDB database, this example creates a **QueryDef** with parameters and then lists those parameters in a **Snapshot**.

```
Dim ListSet As Snapshot, MyDB As Database, MyQuery As QueryDef
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("Large Orders") ' Create QueryDef.
MyQuery.SQL = "Parameters FindAmount Currency, FindFreight Currency; SELECT *
➥ FROM Orders WHERE [Order Amount] > FindAmount;"
MyQuery.FindAmount = 1000
Set ListSet = MyQuery.ListParameters()        ' Copy parameters to ListSet.
...
MyDB.DeleteQueryDef("Large Orders")           ' Delete QueryDef.
```

`ListSet` looks like the following table.

Name	Type
FindAmount	5
FindFreight	5

ListRows Property

Applies To Tool and control (combo box).

Description Specifies the maximum number of rows to display in the list portion of a combo box in Form view and Datasheet view.

Setting Enter an integer for the maximum number of rows. The default setting is 8.

Remarks If the actual number of rows exceeds the number in the ListRows setting, a vertical scroll bar appears in the combo box.

Access Basic Use a numeric expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also Height, Width Properties; ListWidth Property

ListTables Method

Description Creates a **Snapshot** with one record for each table or **QueryDef** in a specified database.

Syntax **Set** *snapshot* = *database*.**ListTables**( )

Remarks The **ListTables** method uses the following arguments.

Argument	Description
snapshot	Name of the **Snapshot** you want to create
database	Name of an open database object that contains the tables and **QueryDefs** you want to list

The following table shows the **Snapshot** fields.

Field	Data type	Description
Name	String	Name of the table or **QueryDef.**
DateCreated	Date/Time	Date on which the table or **QueryDef** was created.
LastUpdated	Date/Time	Date on which the table or **QueryDef** definition was last changed.

Field	Data type	Description
TableType	Long	One of the intrinsic constants that identifies the kind of table or **QueryDef**: DB_TABLE (native table) DB_ATTACHEDTABLE (attached table) DB_ATTACHEDODBC (attached ODBC table) DB_QUERYDEF (**QueryDef**)
RecordCount	Long	Number of records in the table; if TableType is DB_QUERYDEF, RecordCount is 0.
Attributes	Long	Table flags.

Note When you use the **ListTables** method to create a **Snapshot**, you can inspect the Attributes field in the **Snapshot** using the Attributes constants.

The **Snapshot** contains records for the Microsoft Access system tables, as well as for the tables you create. The Microsoft Access system table names begin with the prefix MSys.

You can't use the Find methods on a **Snapshot** created by **ListTables**.

If the specified database contains no user-defined tables and no **QueryDefs**, the **Snapshot** will include only the records of the system tables.

See Also

ListFields Method

Example

This example creates a **Snapshot** that lists all of the tables in the current database.

```
Dim MyDB As Database, ListSet As Snapshot
Set MyDB = CurrentDB()
Set ListSet = MyDB.ListTables()          ' Copy table information to ListSet.
```

Using this example with the NWIND.MDB database might produce the following table.

Name	DateCreated	LastUpdated	RecordCount	Attributes
Categories	11/22/91 09:22:39	11/22/91 09:22:45	8	0
Customers	11/22/91 09:22:50	11/22/91 09:22:54	90	0
Employees	11/22/91 09:22:58	11/22/91 09:23:04	9	0
MSysACEs	12:00:00	12:00:00	17	–2147483648
MSysColumns	12:00:00	12:00:00	17	–2147483648
MSysForms	11/22/91 09:20:45	01/22/92 01:33:39	17	2
MSysIndexes	12:00:00	12:00:00	17	–2147483648
MSysObjects	12:00:00	12:00:00	17	–2147483648
MSysQueries	12:00:00	12:00:00	17	–2147483648
MSysRefs	11/22/91 10:27:31	11/22/91 10:27:33	17	2
MSysScripts	11/22/91 09:20:56	11/22/91 09:20:58	17	2
Order Details	01/21/92 01:09:12	02/04/92 05:48:41	17	0
Orders	11/22/91 09:23:38	11/22/91 10:28:24	1078	0

Name	DateCreated	LastUpdated	RecordCount	Attributes
Products	11/22/91 09:24:00	11/22/91 09:24:04	77	0
Shippers	11/22/91 09:24:09	11/22/91 09:24:10	3	0
Suppliers	11/22/91 09:24:17	11/22/91 09:24:20	29	0

ListWidth Property

Applies To Tool and control (combo box).

Description Specifies the width of the drop-down list box portion of a combo box.

Setting Enter the desired width in the current unit of measurement. To use a unit other than the default, include a measurement indicator, such as "cm" or "in." The drop-down list can be wider than the text box but can't be narrower. The Auto setting makes the drop-down list box the same width as the combo box.

Remarks If you want a multiple-column drop-down list, you can change the Auto default setting to make the drop-down list wide enough to fit all the columns.

Tip When designing combo boxes, be sure to leave space for a scroll bar.

Access Basic Use a numeric expression to set the value of this property. You can also use a string expression that includes a unit of measurement, such as `"1.5 in"` or `"2 cm"`, or `"Auto"` to set the width to Auto.

Tool properties aren't available from Access Basic.

Design view: read/write. Other views: read-only.

See Also Height, Width Properties; ListRows Property

Loc Function

Description Returns the current position within an open file.

Syntax **Loc**(*filenumber*)

Remarks The argument *filenumber* is the number used in the **Open** statement to open the file. You can use any numeric expression that evaluates to the number of an open file. For **Random** files, the **Loc** function returns the number of the last variable read from or written to the

file. For sequential files, **Loc** returns the current byte position in the file divided by 128. For **Binary** mode files, **Loc** returns the position of the last byte read or written.

See Also **EOF** Function, **LOF** Function, **Open** Statement

Example This example uses the **Loc** function to report the current position within an open data file.

```
Open "TESTFILE" For Input As #1                    ' Open file just created.
For I = 1 To 50
  Input #1, Tmp                                    ' Read some data, discard.
  MsgBox "Current file position = " & Loc(1)       ' Get current position.
Next I
Close #1                                           ' Close file.
```

Lock, Unlock Statements

Description Control access by other processes to all or part of an opened file.

Syntax **Lock** [#]*filenumber* [,{*record* | [*start*] **To** *end*}]
. . .
Unlock [#]*filenumber* [,{*record* | [*start*] **To** *end*}]

Remarks The **Lock** and **Unlock** statements are used in networked environments in which several processes might need access to the same file. The statements use these arguments.

Argument	Description
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.
record	Number of the record or byte to lock. It can be any number from 1 through 2,147,483,647 (equivalent to $2^{31}-1$). A record can be up to 65,535 bytes long.
start	Number of the first record or byte to lock.
end	Number of the last record or byte to lock.

For **Binary** mode files, *record*, *start*, and *end* represent the number of a byte relative to the beginning of a file. The first byte in a file is byte 1.

For **Random** files, *record*, *start*, and *end* represent the number of a record relative to the beginning of a file. The first record is record 1.

If the file has been opened for sequential input or output, **Lock** and **Unlock** affect the entire file, regardless of the range specified by *start* and *end*.

Lock and **Unlock** statements are always used in pairs. The arguments to **Lock** and **Unlock** must match exactly.

If you specify just one record, then only that record is locked or unlocked. If you specify a range of records and omit a starting record (*start*), all records from the first record to the end of the range (*end*) are locked or unlocked. Using **Lock** without *record* locks the entire file; using **Unlock** without *record* unlocks it.

Lock and **Unlock** execute only at run time. If you are using a version of MS-DOS that supports networking (MS-DOS version 3.1 or later), you must run the SHARE.EXE program to enable locking operations. You can't use **Lock** and **Unlock** with earlier versions of MS-DOS.

Caution Be sure to remove all locks with an **Unlock** statement before closing a file or terminating your program. Failure to remove locks produces unpredictable results. The arguments to **Lock** and **Unlock** must match exactly.

See Also **Open** Statement

Example This example illustrates the use of the **Lock** and **Unlock** statements. While a record (`RecNum`) is being updated, access to it on a network is locked to other users until the update is complete.

```
Open "TESTFILE" For Random Shared As #1 Len = Len(CustRec)
Do
  Lock #1, RecNum                          ' Lock the current record.
  Get #1, RecNum, CustRec                  ' Read a record.
  ...                                      ' Do something that
  ...                                      '  changes the data.
  Put #1, RecNum, CustRec                  ' Write the record back out.
  Unlock #1, RecNum                        ' Unlock the record.
Loop Until Done = True                     ' Loop until done.
Close #1                                   ' Close the file.
```

Locked Property

See Enabled, Locked Properties

LockEdits Property

Applies To **Tables**, **Dynasets**.

Description Sets the locking condition that will be in effect during editing.

Setting The LockEdits property settings are:

Setting	Description
–1	(Default) Pessimistic locking is in effect for all editing. The page containing the record you are editing is locked as soon as you use the **Edit** method. The page is unlocked when you use the **Update** method.
0	Optimistic locking is in effect for all editing. The page containing the record is locked only while the record is being updated (with the **Update** method).

Usage Read/write.

Remarks If a page is locked (LockEdits is **True** (–1)), no other user can edit records on the same page. If you set LockEdits to **True** and another user already has the page locked, an error occurs when you use the **Edit** method.

If you set LockEdits to **False** (0) and later use **Update** while the page is locked by another user, an error occurs.

See Also **Edit** Method, **Update** Method

LOF Function

Description Returns the size of an open file in bytes.

Syntax **LOF**(*filenumber*)

Remarks The argument *filenumber* is the number used in the **Open** statement to open the file. You can use any numeric expression that evaluates to the number of an open file.

LOF can be used only on disk files.

See Also **EOF** Function, **Loc** Function, **Open** Statement

Example This example uses **LOF** to determine the size of an open disk file.

```
Open "TESTFILE" For Input As #1          ' Open file.
FileLength = LOF(1)                      ' Get length of file.
Close #1                                 ' Close file.
```

Log Function

Description Returns the natural logarithm of a number.

Syntax **Log**(*number*)

Remarks The argument *number* can be any valid numeric expression that results in a value greater than 0. The natural logarithm is the logarithm to the base *e*. The constant *e* approximately equals 2.718282.

You can calculate base-*n* logarithms for any number *x* by dividing the natural logarithm of *x* by the natural logarithm of *n* as follows:

Log*n* = **Log**(*x*) / **Log**(*n*)

The following example illustrates a **Function** procedure that calculates base-10 logarithms:

```
Static Function Log10(X)
  Log10 = Log(X) / Log(10#)
End Function
```

See Also **Exp** Function

Example This example uses **Log** to calculate the natural logarithm of *e* to the third power. The result is 3.

```
= Log(Exp(1) ^ 3)
```

Long Data Type

Description **Long** (long integer) variables are stored as signed 32-bit numbers (4 bytes) ranging in value from –2,147,483,648 to 2,147,483,647. The type-declaration character for a **Long** is **&** (ANSI character 38).

See Also Access Basic Data Types, **Integer** Data Type

LSet Statement

Description Left aligns a string within the space of a string variable or copies a variable of one user-defined type to another variable of a different user-defined type.

Syntax 1 **LSet** *stringvar* = *stringexpr*

Syntax 2 **LSet** variable1 = variable2

Remarks The **LSet** statement uses these arguments.

Argument	Description
stringvar	Name of a **String** variable
stringexpr	String expression to be left aligned within *stringvar*
variable1	Name of the user-defined type variable being copied to
variable2	Name of the user-defined type variable being copied from

If *stringexpr* is shorter than *stringvar*, **LSet** left aligns *stringexpr* within *stringvar*. **LSet** replaces any leftover characters in *stringvar* with spaces.

If *stringexpr* is longer than *stringvar*, **LSet** places only the leftmost characters, up to the length of the *stringvar*, in *stringvar*. Characters beyond the length of *stringvar* are truncated from the right.

Use **LSet** with Syntax 2 to assign one user-defined type variable to another. The following example copies the contents of `RecTwo` (a user-defined type variable) to `RecOne` (a variable of another user-defined type):

```
Type TwoString
  StrFld As String * 2
End Type
Type ThreeString
  StrFld As String * 3
End Type
Dim RecOne As TwoString, RecTwo As ThreeString
LSet RecOne = RecTwo
```

Because `RecOne` is 2 bytes long, only 2 bytes are copied from `RecTwo`. **LSet** copies only the number of bytes in the shorter of the two user-defined type variables.

You can't use **LSet** to copy variables of different user-defined types if either contains a variable-length string or a **Variant**.

See Also **RSet** Statement

Example This example uses **LSet** to left align text in `Text2` (a **String** variable) within `Field1` (another **String** variable).

```
Dim Field1 As String * 25                ' Create a fixed-length variable.
Text2 = "17-character text"              ' Create a 17-character text.
LSet Field1 = Text2                      ' Left align the 17-character text
                                         '  within the 25-character space.
```

This example uses **LSet** to assign `ClientListRec` (a variable of one user-defined type) to `MailListRec` (a variable of another user-defined type).

```
LSet MailListRec = ClientListRec
```

LTrim, LTrim$, RTrim, RTrim$, Trim, Trim$ Functions

Description Return a copy of a string with leading (leftmost), trailing (rightmost), or both leading and trailing spaces removed.

Syntax [**L** | **R**]**Trim**[**$**](*stringexpr*)

Remarks **LTrim**[**$**] removes leading spaces from a string; **RTrim**[**$**] removes trailing spaces. **Trim**[**$**] removes leading and trailing spaces.

LTrim, **RTrim**, and **Trim** each return a **Variant**; **LTrim$**, **RTrim$**, and **Trim$** each return a **String**.

The *stringexpr* argument can be any string expression. However, only **LTrim**, **RTrim,** and **Trim** can accept a **Variant** of **VarType** 1 (**Null**) as *stringexpr*, in which case, a **Null** is returned.

Example This example uses **LTrim**, **RTrim**, and **Trim** to strip leading and trailing spaces from the variable `TestString`.

```
= LTrim(TestString)                      ' Strip leading spaces.
= RTrim(TestString)                      ' Strip trailing spaces.
= Trim(TestString)                       ' Strip leading and trailing spaces.
```

Max Function

See **Min**, **Max** Functions

Maximize Action

Description Enlarges the active window so that it fills the Microsoft Access window.

Remarks Use this action when you want to see as much of the object in the active window as possible.

This action has the same effect as clicking the Maximize button in the window's upper-right corner or choosing the Maximize command from the Control menu.

You can use the Restore action to restore a maximized window to its previous size.

Tip You may need to use the SelectObject action if the window you want to maximize isn't the active window.

See Also Minimize Action, Restore Action, SelectObject Action

Access Basic

Syntax

DoCmd Maximize

Remarks

This action takes no arguments.

Mid, Mid$ Functions

Description Return a string that is part of some other string.

Syntax **Mid**[$](*stringexpr*, *start*[, *length*])

Remarks **Mid** returns a **Variant**; **Mid$** returns a **String**.

The **Mid**[$] function uses these arguments.

Argument	Description
stringexpr	String expression from which another string is created. This can be any string expression. However, only **Mid** can accept a **Variant** of **VarType** 1 (**Null**) as *stringexpr*, in which case, a **Null** is returned.
start	**Long** expression that indicates the character position in *stringexpr* at which the part to be taken begins.
length	**Long** expression that indicates the number of characters to return.

The arguments *start* and *length* must be between 1 and approximately 65,535, inclusive. If *length* is omitted or if there are fewer than *length* characters in the text (including the character at *start*), the **Mid**[$] function returns all characters from the *start* position to the end of the string.

If *start* is greater than the number of characters in *stringexpr*, **Mid**[$] returns a zero-length string.

Use the **Len** function to find the number of characters in *stringexpr*.

See Also **Left**, **Left$** Functions; **Len** Function; **Mid, Mid$** Statements; **Right**, **Right$** Functions

Example In this example, the **Mid** function returns the middle word from a variable containing three words of text.

```
TestStr = "Mid Function Demo"                    ' Create text string.
SpcPos1 = InStr(1, TestStr, Chr(32))             ' Find first space.
SpcPos2 = InStr(SpcPos1 + 1, TestStr, Chr(32))   ' Find next space.
WordLen = (SpcPos2 - SpcPos1) - 1                ' Calculate length of
                                                 '  second word.
MidWord = Mid(TestStr, SpcPos1 + 1, WordLen)     ' Return second word.
```

Mid, Mid$ Statements

Description Replace part of a string with another string.

Syntax **Mid**[$](*stringvar*, *start* [, *length*]) = *stringexpr*

Remarks There is no functional difference between **Mid** and **Mid$**. The **Mid**[$] statement uses these arguments.

Argument	Description
stringvar	**String** or **Variant** (**VarType** 8) variable to modify
start	Character position in *stringvar* where the replacement text begins
length	Number of characters to replace
stringexpr	String expression that replaces part of *stringvar*

The arguments *start* and *length* must be between 1 and approximately 65,535, inclusive. The argument *stringvar* must be a variable, but *stringexpr* can be any string expression.

The optional argument *length* refers to the number of characters from the argument *stringexpr* that are used in the replacement. If *length* is omitted, all of *stringexpr* is used. Whether or not *length* is included, the number of characters replaced is always less than or equal to the number of characters in *stringvar*.

See Also **Mid**, **Mid$** Functions

Example This example uses the **Mid$** statement to substitute `dog` for `fox` within the sample text.

```
TestText = "The quick brown fox jumped over the wall."
Start = InStr(TestText,"fox")              ' Find where fox begins.
Mid$(TestText, Start, 3) = "dog"           ' Replace fox with dog.
```

Min, Max Functions

Description Return the minimum or maximum of a set of values contained in a specified field on a query, form, or report.

Syntax **Min**(*expr*)

Max(*expr*)

Remarks The **Min** and **Max** functions use the following argument.

Argument	Description
expr	String expression identifying the field that contains data you want to evaluate, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

You can use **Min** and **Max** to determine the smallest and largest values in a field, including Text fields. For example, you could use **Min** and **Max** to return the lowest and highest freight cost.

You can use **Min** and **Max** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **DMin**, **DMax** Functions; **Count** Function; **First**, **Last** Functions; **StDev**, **StDevP** Functions; **Sum** Function; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to return the lowest and highest sale for orders shipped to the United Kingdom. You can enter these expressions in the SQL dialog box in the Query window.

```
SELECT Min([Order Amount]) FROM Orders WHERE [Ship Country] = 'UK';
SELECT Max([Order Amount]) FROM Orders WHERE [Ship Country] = 'UK';
```

The next example returns the lowest sale for the underlying records in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition that limits the search to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= Min([Order Amount])
```

Minimize Action

Description Reduces the active window to an icon at the bottom of the Microsoft Access window.

Remarks Use this action to remove a window from the screen while leaving the object open. You can also use this action to open an object without displaying its window. To display the object, use the SelectObject action with either the Maximize or Restore action. The Restore action restores a minimized window to its previous size.

The Minimize action has the same effect as clicking the Minimize button in the window's upper-right corner or choosing the Minimize command from the Control menu.

Tips

- You may first need to use the SelectObject action if the window you want to minimize isn't the active window.
- To minimize the Database window, use the SelectObject action with the In Database Window argument set to Yes and then use the Minimize action.
- You can use the Hide command on the Window menu to hide the active window. Instead of being reduced to an icon, the window becomes invisible. Use the Show command from the Window menu to make the window reappear. Use the DoMenuItem action to carry out either of these commands from a macro.
- You can also use the SetValue action to set a form's Visible property to hide or show the form's window.

See Also DoMenuItem Action, Maximize Action, OpenForm Action, Restore Action, SelectObject Action

Access Basic

Syntax

DoCmd Minimize

Remarks

This action takes no arguments.

Minute Function

Description Returns an integer between 0 and 59, inclusive, that represents the minute of the hour corresponding to the time provided as an argument.

Syntax **Minute**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Hour** Function, **Month** Function, **Now** Function, **Second** Function, **Weekday** Function, **Year** Function

Example In this example, the **TimeValue** function creates a **Variant** of **VarType** 7 (Date) for midnight. The **Hour**, **Minute**, and **Second** functions determine the hour, minute, and second values so the difference can be calculated.

```
Midnight = TimeValue("23:59:59")
Instant = Now                                          ' Get current time.
HourDiff = Hour(Midnight) - Hour(Instant)              ' Get differences.
MinuteDiff = Minute(Midnight) - Minute(Instant)
SecondDiff = Second(Midnight) - Second(Instant) + 1
If SecondDiff = 60 Then
  MinuteDiff = MinuteDiff + 1                          ' Add 1 to minute.
  SecondDiff = 0                                       ' Zero seconds.
End If
If MinuteDiff = 60 Then
  HourDiff = HourDiff + 1                              ' Add 1 to hour.
  MinuteDiff = 0                                       ' Zero minutes.
End If
TotalMinDiff = (HourDiff * 60) + MinuteDiff            ' Get totals.
TotalSecDiff = (TotalMinDiff * 60) + SecondDiff
Msg = "There are a total of " & Format(TotalSecDiff, "#,##0")
Msg = Msg & " seconds until midnight. That translates to "
Msg = Msg & HourDiff & " hours, " & MinuteDiff
Msg = Msg & " minutes, and " & SecondDiff & " seconds."
MsgBox Msg                                             ' Display message.
```

MIRR Function

Description Returns the modified internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax **MIRR**(*valuearray*( ), *financerate*, *reinvestrate*)

Remarks The modified internal rate of return is the internal rate of return when payments and receipts are financed at different rates. The **MIRR** function takes into account both the cost of the investment (*financerate*) and the interest rate received on reinvestment of cash (*reinvestrate*).

The **MIRR** function uses the following arguments.

Argument	Description
valuearray( )	Array of cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).
financerate	Interest rate paid as the cost of financing.
reinvestrate	Interest rate received on gains from cash reinvestment.

The arguments *financerate* and *reinvestrate* are percentages expressed as decimal values. For example, 12 percent is expressed as 0.12.

The **MIRR** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence.

Because the **MIRR** function requires an array of cash flows, it can't be used as an expression on a form. It can be used only in Access Basic.

See Also **IRR** Function, **Rate** Function

Example This example returns the modified internal rate of return for a series of cash flows contained in the array `Values()`. `LoanAPR` represents the financing interest, and `InvAPR` represents the interest rate received on reinvestment.

```
Static Values(5) As Double                    ' Set up array.
LoanAPR = .1                                  ' Loan rate.
InvAPR = .12                                  ' Reinvestment rate.
Fmt = "#0.00"                                 ' Define money format.
Values(0) = -70000                            ' Business start-up costs.
Values(1) = 22000 : Values(2) = 25000         ' Positive cash flows reflecting
Values(3) = 28000 : Values(4) = 31000      ' income for four successive years.
RetRate = MIRR(Values(), LoanAPR, InvAPR)     ' Calculate internal rate.
Msg = "The modified internal rate of return for these five cash flows is "
Msg = Msg & Format(Abs(RetRate) * 100, Fmt) & "%."
MsgBox Msg                                    ' Display internal return rate.
```

MkDir Statement

Description Creates a new directory.

Syntax **MkDir** *pathname*

Remarks The argument *pathname* is a string expression that specifies the name of the directory to be created. This argument must contain fewer than 128 characters and has the following syntax:

[*drive*:] [\ ]*directory*[*directory*] . . .

The argument *drive* is an optional drive specification; the argument *directory* is a directory name.

The **MkDir** statement works like the operating system command **mkdir**.

If you use **MkDir** to create a directory whose name contains an embedded space, you may be able to access it with some applications, but you can't remove it using standard operating system commands. To remove such a directory, use the **RmDir** statement from within Access Basic.

See Also **ChDir** Statement; **CurDir**, **CurDir$** Functions; **RmDir** Statement

Example This example uses the **MkDir** statement to create a \TMP subdirectory off the root directory of the currently logged drive. The **RmDir** statement removes it at the user's request.

```
On Error Resume Next                        ' Set up error handler.
CurDrv = Left(CurDir, 2)                    ' Get current drive letter.
TmpPath = UCase(CurDrv + "\tmp")            ' Make path specification.
MkDir TmpPath                               ' Make new directory.
If Err = 75 Then                            ' Check if directory exists.
  Msg = TmpPath & " directory already exists."
Else
  Msg = TmpPath & " directory created."
End If
Msg = Msg & " Do you want it removed?"
Answer = MsgBox(Msg, 4)                     ' Display message and get
If Answer <> 7 Then RmDir TmpPath           ' user response.
```

Mod Operator

Description Divides two numbers and returns only the remainder.

Syntax *result* = *operand1* **Mod** *operand2*

Remarks The modulus, or remainder, operator divides *operand1* by *operand2* (rounding floating-point numbers to integers) and returns only the remainder as *result*. For example, in the expression `A = 19 Mod 6.7, A` (which is *result*) equals 5. The operands can be any numeric expression.

Usually, the data type of *result* is an **Integer**, a **Long**, or a **Variant** of **VarType** 2 (**Integer**) or **VarType** 3 (**Long**). However, *result* is a **Null** (**VarType** 1) if one or both operands are **Null** expressions. Any operand that is Empty (**VarType** 0) is treated as 0.

See Also & Operator, * Operator, + Operator, – Operator, / Operator, \ Operator, ^ Operator, Comparison Operators, Operator Precedence, **VarType** Function

Example This example determines the remainder after 100,000 units are divided among seven people using integer division and returns 5.

```
= 100000 Mod 7
```

Modal Property

Applies To Forms.

Description Determines whether a form opens as a modal form, which retains the focus until it is closed.

Setting The Modal property settings are:

Setting	Description
Yes	The form opens as a modal form in Form view.
No	(Default) The form isn't a modal form.

Remarks When you open a modal form, other windows are disabled until you close it. To disable the menus and tool bar in addition to the windows, set both the Modal property and the PopUp property to Yes.

You can use the Modal and PopUp properties to create a dialog box.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also PopUp Property

Month Function

Description Returns an integer between 1 and 12, inclusive, that represents the month of the year for a date argument.

Syntax **Month**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Hour** Function, **Minute** Function, **Now** Function, **Second** Function, **Weekday** Function, **Year** Function

Example This example uses the **Month** function to determine the month of the year.

```
Serial = Now                                    ' Get date serial number.
MonthNum = Month(Serial)                        ' Get current month number.
Select Case MonthNum                            ' Get proper suffix
  Case 1                                        '  for each number.
    MonthSuffix = "st"
  Case 2
    MonthSuffix = "nd"
  Case 3
    MonthSuffix = "rd"
  Case Else
    MonthSuffix = "th"
End Select
Msg = Format(Serial, "mmmm") & " is the " & MonthNum
Msg = Msg & MonthSuffix & " month of the year."
MsgBox Msg                                      ' Display message.
```

MoveFirst, MoveLast, MoveNext, MovePrevious Methods

Description Locate the first, last, next, or previous record in a specified recordset and makes that record the current record.

Syntax

recordset.**MoveFirst**

recordset.**MoveLast**

recordset.**MoveNext**

recordset.**MovePrevious**

Remarks The Move methods use the following argument.

Argument	Description
recordset	Name of an open recordset

Use the Move methods to move from record to record without applying a condition. Use the Find methods to locate records in a **Dynaset** or **Snapshot** that satisfy a certain condition. To locate a record in a **Table**, use the **Seek** method.

Caution If you edit the current record, be sure you save the changes with the **Update** method before you move to another record. If you move without updating, your changes will be lost, and no error will occur.

When you open or create *recordset*, the first record is current and the BOF property is **False** (0). If *recordset* contains no records, the BOF property is **True** (–1), and there is no current record.

If the first or last record is already current when you use **MoveFirst** or **MoveLast**, the current record doesn't change.

If you use **MovePrevious** when the first record is current, the BOF property will be **True** and there will be no current record. If you use **MovePrevious** again, an error occurs; BOF remains **True**.

If you use **MoveNext** when the last record is current, the EOF property will be **True** and there will be no current record. If you use **MoveNext** again, an error occurs; EOF remains **True**.

If *recordset* refers to a **Table**, movement follows the **Table's** current index. You can set the current index with the Index property. If you don't set the current index, movement follows the order in which the records were added to the **Table**.

If you use **MoveLast** during the execution of a **QueryDef**, the query will be forced to completion.

If *recordset* doesn't refer to an open recordset, an error occurs.

See Also BOF Property; EOF Property; **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods; Index Property; RecordCount Property; **Seek** Method; **Update** Method

Example This example changes the job title of all sales representatives in the Employees table of the NWIND.MDB database. After opening the table, it uses **MoveFirst** to locate the first record and **MoveNext** to move to the next record. For each record satisfying the title condition, it changes the title and saves the change with **Update**.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Employees")          ' Open Table.
MyTable.Index = "Last Name"                        ' Set current index.
...
MyTable.MoveFirst                                  ' Locate first record.
Do Until MyTable.EOF                               ' Begin loop.
  If MyTable!Title = "Sales Representative" Then' Check title.
    MyTable.Edit                                   ' Enable editing.
    MyTable!Title = "Account Executive"            ' Change title.
    MyTable.Update                                 ' Save changes.
  End If
  MyTable.MoveNext                                 ' Locate next record.
Loop                                               ' End of loop.
MyTable.Close                                      ' Close Table.
```

Tips

- You could change the records in reverse order by moving to the last record with **MoveLast** and then using **MovePrevious** instead of **MoveNext**.
- Using an update query to change job titles might be more efficient.

MoveLast Method

See **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods

MoveLayout, NextRecord, PrintSection Properties

Apply To Report sections.

Description Use these properties when you preview or print a report or save the output to a file.

- MoveLayout—specifies whether Microsoft Access should move to the next printing location on the page.
- NextRecord—specifies whether a section should advance to the next record.
- PrintSection—specifies whether a section should be printed on the page.

After each section, each property is reset to True.

Setting To set these properties, specify a macro or Access Basic function for the section's OnFormat property. If you use the SetValue action in a macro, do not include the Section property to set the Item argument. For example, to refer to a section's MoveLayout property, use Report.**MoveLayout**.

In a function, use the following syntax:

Reports!*reportname*.[**MoveLayout** | **NextRecord** | **PrintSection**] = [True | False]

These properties are automatically set to True before the OnFormat event.

The MoveLayout property settings are:

Setting	Description
True (nonzero)	(Default) The section's Left and Top properties are advanced to the next print location.
False (0)	The section's Left and Top properties are unchanged.

The NextRecord property settings are:

Setting	Description
True (nonzero)	(Default) The data advances to the next record.
False (0)	The data doesn't advance to the next record.

The PrintSection property settings are:

Setting	Description
True (nonzero)	(Default) The section is printed on the page.
False (0)	The section isn't printed on the page.

Usage Design view: not available. Other views: read/write.

Remarks These properties are useful when you want to use a report as a template into which you write data from a macro or Access Basic as you print.

The following table shows the possible settings for these properties.

MoveLayout	NextRecord	PrintSection	Description
True	True	True	(Default) Move to the next print location, get the next record, and print it.
True	False	True	Move to the next print location, don't advance to the next record, but do print the section. This combination is typically used when the data in a section requires more space than the layout allows and you want to print the remaining data in the space that would be occupied by the next section.
False	True	False	Skip a record without leaving a blank space on the page.
True	True	False	Skip a record and leave a blank space on the page.
True	False	False	Leave a blank space without skipping a record, as in a sparse matrix.
False	True	True	Print the current record on top of the last record as an overlay.
False	False	True	Not allowed.
False	False	False	Not allowed.

See Also NewRowOrCol Property

MoveNext Method

See **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods

MovePrevious Method

See **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods

MoveSize Action

Description Moves or resizes the active window.

Action argument	Description
Right	The new horizontal position of the window's upper-left corner, measured from the left edge of its containing window
Down	The new vertical position of the window's upper-left corner, measured from the top edge of its containing window
Width	The window's new width
Height	The window's new height

Note Each measurement is in inches or centimeters, depending on the units set in the International section of the Microsoft Windows Control Panel.

Remarks If you leave an argument blank, Microsoft Access uses the window's current setting.

You should enter a value for at least one argument.

This action is similar to choosing the Move or Size command from the window's Control menu. With the menu commands, you use the keyboard's arrow keys to move or resize the window. With the MoveSize action, you enter the position and size measurements directly.

You can use this action in Design view, Form view, and Datasheet view.

Tips

- To move a window without resizing it, enter values for Right and Down but leave Width and Height blank.
- To resize a window without moving it, enter values for Width and Height but leave Right and Down blank.

See Also Restore Action, SelectObject Action

Access Basic

Syntax

DoCmd MoveSize [*right*] [, *down*] [, *height*] [, *width*]

Argument	Description
right	Numeric expression
down	Numeric expression
height	Numeric expression
width	Numeric expression

Remarks

You must include at least one argument for the MoveSize action. If you leave an argument blank, the current setting for the window is used.

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

The units for the arguments are twips.

Example

This example moves the active window and changes its width but not its height.

```
DoCmd MoveSize 1440, 2400, , 2000
```

MsgBox Action

Description Displays a message box containing a warning or an informational message.

Action argument	Description
Message	The text in the message box. You can type up to 255 characters or enter an expression (preceded by an equal sign).
Beep	Specifies whether your computer's speaker sounds a beep tone once when the message displays. Select Yes to sound the speaker once. Select No if you don't want a beep. The default is Yes.
Type	The type of message box. Each type has a different icon (see the table below). Select None, Critical, Warning?, Warning!, or Information. The default is None.
Title	The text displayed in the message box title bar. For example, "Customer ID Validation." If you leave this argument blank, "Microsoft Access" is displayed.

The Type argument settings and their icon styles are:

Type	Icon style
None	No icon
Critical	
Warning?	
Warning!	
Information	

Remarks Often, you'll use the MsgBox action with validation macros. When a field or record fails a validation condition in the macro, a message box can display an error message and provide instructions about the kind of data that should be entered.

See Also CancelEvent Action; **MsgBox** Function, **MsgBox** Statement

Access Basic This action isn't available from Access Basic. Use the **MsgBox** statement or function.

MsgBox Function, MsgBox Statement

Description Display a message in a dialog box and wait for the user to choose a button. The **MsgBox** function returns a value indicating which button the user has chosen; the **MsgBox** statement does not.

Syntax Function: **MsgBox**(*msg* [, *type* [, *title*]])

Statement: **MsgBox** *msg* [, *type* [, *title*]]

Remarks The **MsgBox** function and statement both use these arguments.

Argument	Description
msg	String expression displayed as the message in the dialog box.
type	Numeric expression that controls the buttons and icons to display in the dialog box.
title	String expression displayed in the title bar of the dialog box.

MsgBox displays a maximum of 1024 characters. Longer messages are truncated after the 1024th character. Message strings longer than 255 characters with no intervening spaces are truncated after the 255th character.

MsgBox breaks lines automatically at the right edge of the dialog box. If you want to set line breaks yourself, place a carriage return (ANSI character 13) and a linefeed (ANSI character 10) before the first character of the text that is to begin each new line.

The argument *type* is the sum of values that specify the number and type of buttons to display, the icon style to use, and the identity of the default button. The following tables illustrate the values used and the meaning of each group of values.

Number and type of buttons:

Value	Meaning
0	Display OK button only.
1	Display OK and Cancel buttons.
2	Display Abort, Retry, and Ignore buttons.
3	Display Yes, No, and Cancel buttons.
4	Display Yes and No buttons.
5	Display Retry and Cancel buttons.

Icon style:

Value	Icon	Meaning
0		Display no icon.
16		Display Critical Message icon.
32		Display Warning Query icon.
48		Display Warning Message icon.
64		Display Information Message icon.

Default button:

Value	Meaning
0	First button is default.
256	Second button is default.
512	Third button is default.

The first group of values (1–5) describes the number and type of buttons displayed in the dialog box; the second group (0, 16, 32, 48, 64) describes the icon style; and the third group (0, 256, 512) determines which button is the default. When adding numbers to create a final value for the argument *type*, use only one number from each group.

If you omit *type*, **MsgBox** displays a single OK button in the dialog box and makes it the default button; no icon is displayed. Generally, displaying more than one button with the **MsgBox** statement is not useful because statements don't return values.

The value returned by the **MsgBox** function indicates which button has been pressed, as shown in the following table.

Value	Button pressed
1	OK
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

If the dialog box displays a Cancel button, pressing the ESC key has the same effect as choosing Cancel.

If you omit the argument *title*, **MsgBox** uses "Microsoft Access" as the default title.

See Also **InputBox**, **InputBox$** Functions

Example This example uses **MsgBox** to display a critical-error message in a dialog box with a Yes button and a No button. The No button is the default response. The **MsgBox** function returns a value based on the button chosen by the user. The **MsgBox** statement uses that value to display a message that indicates which button was chosen.

```
Const MB_OK = 0, MB_OKCANCEL = 1                        ' Define buttons.
Const MB_YESNOCANCEL = 3, MB_YESNO = 4
Const MB_ICONSTOP = 16, MB_ICONQUESTION = 32            ' Define Icons.
Const MB_ICONEXCLAMATION = 48, MB_ICONINFORMATION = 64
Const MB_DEFBUTTON2 = 256, IDYES = 6, IDNO = 7          ' Define other.
Title = "MsgBox Demo"
' Put together a sample message box with all the proper components.
Msg = "This is a sample of a critical-error message."
Msg = Msg & " Do you want to continue?"
DgDef = MB_YESNO + MB_ICONSTOP + MB_DEFBUTTON2          ' Describe dialog.
Response = MsgBox(Msg, DgDef, Title)                    ' Get user response.
If Response = IDYES Then                                ' Evaluate response
  Msg = "You chose Yes."                                ' and take appropriate
Else                                                    ' action.
  Msg = "You chose No or pressed Enter."
End If
MsgBox Msg                                      ' Display action taken.
```

MsgBox Statement

See **MsgBox** Function, **MsgBox** Statement

Name Statement

Description Changes the name of a disk file or directory.

Syntax **Name** *oldfilespec* **As** *newfilespec*

Remarks The **Name** statement is similar to the operating system **rename** command, but **Name** can also be used to change the name of a directory. Using **Name**, you can move a file from one directory to another, but you can't move a directory.

The arguments *oldfilespec* and *newfilespec* are string expressions, each of which contains a file name and an optional path. If the path in *newfilespec* exists and is different from the path in *oldfilespec*, the **Name** statement moves the file to the new directory and renames the file if necessary. If *newfilespec* and *oldfilespec* have different paths and the same file name, **Name** moves the file to the new directory and leaves the file name unchanged.

The file specified by *oldfilespec* must exist and the file specified by *newfilespec* cannot already exist. Both *newfilespec* and *oldfilespec* must be on the same drive.

Using **Name** on a file currently open by Access Basic produces an error. You must close an open file before renaming it.

See Also **Kill** Statement

Example This example moves a file from one directory to another and renames it at the same time.

```
FName1 = "TEST1"                          ' Define file names.
FName2 = "TEST2"
Name FName1 As "\TEMP\" & FName2          ' Move and rename.
```

NewRowOrCol Property

Applies To Form and report sections. Ignored for page headers and footers.

Description Specifies whether a section in a multiple-column layout always starts printing at the beginning of a new row or column. If you select Horizontal for Item Layout in the Print Setup dialog box, the section starts a new row; if you select Vertical, the section starts a new column.

Setting The NewRowOrCol property settings are:

Setting	Description
None	(Default) The row or column breaks are determined by the Print Setup dialog box and by available space on the page.
Before Section	The current section, such as a group header section, starts printing in a new row or column. The next section, such as a detail section, starts printing in that same row or column.
After Section	The current section, such as a group header section, starts printing in the current row or column. The next section, such as a detail section, starts printing in the next row or column.
Before & After	The current section starts printing in a new row or column. The next section starts printing in the next row or column.

Remarks Microsoft Access prints forms and reports one section at a time. For example, it prints a report header section first and a group header section next; then it may print a detail section many times, once for each set of data belonging to the group.

You can print the sections of a form or report in multiple columns across a page. The options you select in the Print Setup dialog box control whether the sections print vertically or horizontally within multiple columns. Use the NewRowOrCol property to control the placement of a section among the columns and rows on the printed page. If Print Setup indicates a vertical orientation, this property refers to a column; if it indicates a horizontal orientation, this property refers to a row.

For example, to start a new column before printing the group header section in a multiple-column report with a vertical print orientation, you would set this property to Before Section.

To see a report with two different printing layouts, see the example.

Access Basic

The property settings and their values are:

Setting	Value
None	0
Before Section	1
After Section	2
Before & After	3

Design view: read/write. Other views: read-only.

See Also

ForceNewPage Property, OnFormat Property, OnPrint Property

Example

This example presents two printing layouts for a report that groups the data into four groups (Head1 to Head4). Each group includes three to six records, and each record has field a and field b. The layouts differ only in the Item Layout setting in the Print Setup dialog box and the value of the NewRowOrCol property.

For both layouts, Print Setup Columns = 4.

- Item Layout = Vertical
- NewRowOrCol property on group header section = Before

Head1	Head2	Head3	Head4
1a 1b	1a 1b	1a 1b	1a 1b
2a 2b	2a 2b	2a 2b	2a 2b
3a 3b	3a 3b	3a 3b	3a 3b
4a 4b	4a 4b		4a 4b
5a 5b			5a 5b
			6a 6b

- Item Layout = Horizontal
- NewRowOrCol property on group header section = Both

Head1			
1a 1b	2a 2b	3a 3b	4a 4b
5a 5b			
Head2			
1a 1b	2a 2b	3a 3b	4a 4b
Head3			
1a 1b	2a 2b	3a 3b	
Head4			
1a 1b	2a 2b	3a 3b	4a 4b
5a 5b	6a 6b		

NextRecord Property

See MoveLayout, NextRecord, PrintSection Properties

NoMatch Property

Applies To Recordsets.

Description Indicates whether the application of a Find method or the **Seek** method was successful.

Setting The NoMatch property settings are:

Setting	Description
–1	Application of a Find method or the **Seek** method failed (the desired record wasn't found).
0	Application of a Find method or the **Seek** method was successful (the desired record was found).

Usage Read-only.

Remarks When you first open or create the recordset, NoMatch is **False** (0).

To locate a record, use the **Seek** method on a **Table** or one of the Find methods on a **Dynaset** or **Snapshot**. Then, inspect the NoMatch property setting to see whether the record was found.

If the **Seek** or Find operation fails (no matching record was found), the NoMatch property setting will be **True** (–1); there will be no current record.

If a matching record is found, the setting will be **False**.

See Also BOF Property; EOF Property; **FindFirst, FindLast, FindNext, FindPrevious** Methods; **Seek** Method

Example This example changes the job title of all sales representatives in the Employees table in the NWIND.MDB database. It creates a **Dynaset** and then uses **FindFirst** and **FindNext** to locate every record satisfying the title condition. It prepares each record for editing, changes the title, and saves the change with **Update**.

```
Dim Criteria As String, MyDB As Database, MySet As Dynaset
Criteria = "Title = 'Sales Representative'"      ' Define search criteria.
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Employees")      ' Create Dynaset.
MySet.FindFirst Criteria                 ' Locate first occurrence.
Do Until MySet.NoMatch                   ' Loop until no matching records.
  MySet.Edit                                     ' Enable editing.
  MySet!Title = "Account Executive"              ' Change title.
  MySet.Update                                   ' Save changes.
  MySet.FindNext Criteria                        ' Locate next record.
Loop                                             ' End of loop.
```

Tip Using an update query to change job titles might be more efficient.

Not Operator

Description Used to negate a numeric expression.

Syntax *result* = **Not** *expr*

Remarks The following table illustrates how *result* is determined.

If *expr* is	Then *result* is
true (nonzero)	**False** (0)
false (0)	**True** (–1)
Null	**Null**

The **Not** operator performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

Bit in *expr*	Bit in *result*
0	1
1	0

The **Not** operator inverts the bit values of any variable. If an integer variable has the value 0 (**False**), the variable becomes –1 (**True**); if it has the value –1, it becomes 0. However, a bit-wise **Not** comparison can be performed only in Access Basic code.

See Also **And** Operator, **Eqv** Operator, **Imp** Operator, Operator Precedence, **Or** Operator, **Xor** Operator

Example This example prints a message that depends on the value of variables `A` and `B`, assuming neither variable is a **Null**. If `A` = 10 and `B` = 8, the **Not** expression evaluates **True** because `A` is not equal to `B`.

```
If Not A = B Then
  Debug.Print "A and B aren't equal."
Else
  Debug.Print "A and B are equal."
End If
```

Now Function

Description Returns a date that represents the current date and time according to the setting of the computer's system date and time.

Syntax **Now**[( )]

Remarks When **Now** is used anywhere except in Access Basic code in a Microsoft Access module, the empty parentheses must be included.

The **Now** function returns a **Variant** of **VarType** 7 (Date) containing a date and time that are stored internally as a double-precision number. This number represents a date and time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point represent the date; numbers to the right represent the time.

See Also **Day** Function, **Hour** Function, **Minute** Function, **Month** Function, **Second** Function, **Weekday** Function, **Year** Function

Example In this example, the **Now** function returns the current date as a **Variant** (**VarType** 7) containing a date. The **Format** function formats the date as a **Long** date including the day and date.

```
Today = Now                                    ' Get current date and time.
MsgBox "Today is " & Format(Today, "dddd, mmmm dd, yyyy") & "."
```

NPer Function

Description Returns the number of periods for an annuity based on periodic, constant payments and a constant interest rate.

Syntax **NPer** (*rate*, *pmt*, *pv*, *fv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **NPer** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
pmt	Payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance, is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **FV** Function, **IPmt** Function, **Pmt** Function, **PPmt** Function, **PV** Function, **Rate** Function

Example

This example returns the number of periods over which payments must be made to pay off a loan whose value is contained in `PVal`. Also provided are the interest percentage rate per period (`APR / 12`), the payment (`Payment`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Const ENDPERIOD = 0, BEGINPERIOD = 1          ' When payments are made.
Const MB_YESNO = 4                            ' Define Yes/No buttons.
Const ID_NO = 7                               ' Define No as a response.
FVal = 0                                      ' Usually 0 for a loan.
PVal = InputBox("How much do you want to borrow?")
APR = InputBox("What is the annual percentage rate of your loan?")
If APR > 1 Then APR = APR / 100               ' Ensure proper form.
Payment = InputBox("How much do you want to pay each month?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
TotPmts = NPer(APR / 12, -Payment, PVal, FVal, PayType)
If Int(TotPmts) <> TotPmts Then TotPmts = Int(TotPmts) + 1
MsgBox "It will take you " & TotPmts & " months to pay off your loan."
```

NPV Function

Description

Returns the net present value of an investment based on a series of periodic cash flows (payments and receipts) and a discount rate.

Syntax

NPV(*rate*, *valuearray*( ))

Remarks

The net present value of an investment is the current value of a future series of payments and receipts.

The **NPV** function uses the following arguments.

Argument	Description
rate	Discount rate over the length of the period, expressed as a decimal.
valuearray( )	Array of cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).

The **NPV** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence.

The **NPV** investment begins one period before the date of the first cash flow value and ends with the last cash flow value in the array.

The net present value calculation is based on future cash flows. If your first cash flow occurs at the beginning of the first period, the first value must be added to the value returned by **NPV** and must not be included in the cash flow values of *valuearray*( ).

The **NPV** function differs from the **PV** function in the following ways.

Function	Payments due	Cash flow
NPV	At the end of the period.	Is variable.
PV	At the end or beginning of the period.	Is constant.

Because the **NPV** function requires an array of cash flows, it can't be used as an expression on a form. It can be used only in Access Basic.

See Also **FV** Function, **IRR** Function, **PV** Function

Example This example returns the net present value for a series of cash flows contained in the array `Values()`. `RetRate` represents the fixed internal rate of return.

```
Static Values(5) As Double                  ' Set up array.
Fmt = "###,##0.00"                          ' Define money format.
Guess = .1                                  ' Guess starts at 10%.
RetRate = .0625                             ' Set fixed internal rate.
Values(0) = -70000                          ' Business start-up costs.
Values(1) = 22000 : Values(2) = 25000       ' Positive cash flows reflecting
Values(3) = 28000 : Values(4) = 31000       ' income for four successive years.
NetPVal = NPV(RetRate, Values())            ' Calculate net present value.
Msg = "The net present value of these cash flows is "
Msg = Msg & Format(NetPVal, Fmt) & "."
MsgBox Msg                                  ' Display net present value.
```

Oct, Oct$ Functions

Description Return text that represents the octal value of the decimal argument.

Syntax **Oct**[**$**](*number*)

Remarks **Oct** returns a **Variant**; **Oct$** returns a **String**.

The argument *number* can be any numeric expression. It is rounded to the nearest whole number before being evaluated. An error occurs if *number* is a **Variant** of **VarType** 1 (**Null**).

If the argument is an **Integer** or a **Variant** of **VarType** 2 (**Integer**), up to six octal characters are returned; if the argument is a **Long** or a **Variant** of **VarType** 3 (**Long**), up to 11 octal characters are returned.

You can represent octal numbers directly by preceding numbers in the proper range with `&O`. For example, `&O10` is the octal notation for decimal 8.

See Also **Hex**, **Hex$** Functions

Example This example returns the octal representation of the numeric variable `Num`.

```
= Oct(Num)
```

OldValue Property

Applies To Bound controls on forms.

Description Contains the unedited value of a bound control.

Setting To determine the unedited value of an edited bound control and to assign this value to a variable, use the following syntax:

```
OriginalValue = Forms![MyForm]![MyControl].OldValue
```

Usage Design view: read-only. Other views: read-only.

This property has the same data type as the field to which the control is bound.

Remarks Microsoft Access uses the OldValue property to retrieve the value of a control as it is saved in the record. When you edit a bound control on a form, your changes aren't saved until you save the record. OldValue contains the unedited version of the control.

You can provide your own undo capability by assigning the OldValue property to a control. The following example shows how you can restore the unedited contents of the control `MyControl`:

```
Forms![MyForm]![MyControl] = Forms![MyForm]![MyControl].OldValue
```

If the value of the control hasn't changed, the assignment has no effect. When you save the record, the database is updated and the current value and OldValue will be the same.

Note You can also use transactions to provide additional undo capability.

See Also **Rollback** Statement

Example This example checks to see if a change to a field value is within ten percent of the original value. If the change is greater, it uses OldValue to restore the original value.

```
Sub Validate_Field ()
  Const MB_ICONEXCLAMATION = 48
  Dim NewValue As Currency, OriginalValue As Currency
  Dim CRLF As String, Msg As String
  CRLF = Chr$(13) & Chr$(10)
  NewValue = Forms![My Form]![My Control]
  OriginalValue = Forms![My Form]![My Control].OldValue
  Change = Abs(NewValue - OriginalValue)
  If Change > (OriginalValue * .1) Then
    Msg = "Change is more than ten percent of original."
    Msg = Msg & CRLF & "Restoring original value."
    MsgBox Msg, MB_ICONEXCLAMATION, "Invalid change"
    Forms![My Form]![My Control] = OriginalValue
  End If
End Sub
```

OLEClass Property

Applies To Controls (graph, unbound object frame).

Description Displays a description of the kind of OLE object contained in a graph or an object frame.

Setting This property is read-only.

Remarks Possible descriptions include "Microsoft Excel Chart," "Word Document," and "Paintbrush Picture."

Access Basic The value of this property is a string.

Design view: read-only. Other views: read-only.

On Error Statement

Description Enables an error-handling routine and specifies the location of the routine within a procedure; can be used to disable an error-handling routine.

Syntax **On Error** { **GoTo** *line* | **Resume Next** | **GoTo 0** }

Remarks If you don't use an **On Error** statement, any run-time error that occurs is fatal; that is, Access Basic generates an error message and stops program execution.

The **On Error** statement has these parts.

Part	Description
GoTo *line*	Enables the error-handling routine that starts at *line* (a line label or a line number). Thereafter, if a run-time error occurs, program control branches to *line*. The specified line must be in the same procedure as the **On Error** statement. If it isn't, a compile-time error occurs.
Resume Next	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement in which the error has occurred. In other words, the code continues to execute. You can use the **Err** function in subsequent lines of code to obtain the run-time error number.
GoTo 0	Disables any enabled error handler in the current procedure.

An error handler is enabled when it is referred to by an **On Error GoTo** *line* statement. Once an error handler is enabled, any run-time error causes program control to jump to the enabled error-handling routine and makes the error handler active. An error handler remains active from the time a run-time error has been trapped until a **Resume**, **Exit Sub**, or **Exit Function** statement is executed in the error handler.

If an error occurs while an error handler is active (between the occurrence of the error and the execution of a **Resume**, **Exit Sub**, or **Exit Function** statement), the current procedure's error handler cannot handle the error. If the calling procedure has an enabled error handler, control is returned to the calling procedure and its error handler is activated to handle the error. If the calling procedure's error handler also is active, control is passed back through any previous calling procedures until an inactive error handler is found. If no inactive error handler is found, the error is fatal at the point at which it actually occurred. Each time the error handler passes control back to the calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, program execution resumes in the current procedure at the point designated by the **Resume** statement.

Notice that an error-handling routine is not a **Sub** or **Function** procedure. It is a block of code marked by a line label or line number.

Error-handling routines rely on the value in **Err** to determine the cause of the error. The error-handling routine should test or save this value before any other error can occur or before a procedure that could cause an error is called. The value in **Err** reflects only the most recent error. You can use the **Error[$]** function to return the error message associated with any given run-time error number returned by **Err**.

On Error Resume Next causes program execution to continue with the statement immediately following the statement that caused the run-time error. This allows your program to continue despite a run-time error and then check for the cause of the error. This also lets you build the error-handling routine in line with the procedure rather than transferring control to another location within the procedure.

On Error GoTo 0 disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an **On Error GoTo 0** statement, an error handler is automatically disabled when a procedure is exited.

To prevent error-handling code from executing when no error has occurred, place an **Exit Sub** or **Exit Function** statement immediately ahead of the error-handling routine, as in the following example:

```
Sub InitializeMatrix(Var1, Var2, Var3, Var4)
  On Error GoTo ErrorHandler
  ...
  Exit Sub
ErrorHandler:
  ...
  Resume Next
End Sub
```

Here, the error-handling code follows the **Exit Sub** statement and precedes the **End Sub** statement to partition it from the normal execution flow of the procedure. This is by no means the only solution. Error-handling code can be placed anywhere in a procedure.

See Also **Err, Erl** Functions; **Error, Error$** Functions; **Resume** Statement

Example This example uses the **On Error** statement to set up error handling in a procedure, and **Resume Next** terminates the error-handling routine.

```
Sub ErrorHandlerDemo ()
On Error GoTo ErrorHandler                    ' Set up error handler.
Drive = Chr(Int((26) * Rnd) + Asc("A"))       ' Generate random letter.
Open Drive & ":\TEST\X.DAT" For Input As #1    ' Try to open file.
Close #1                                      ' Close file.
Exit Sub                                      ' Exit before entering.
ErrorHandler:                                 ' Error-handler line
                                              '  label.
```

```
Select Case Err
  Case 53 : MsgBox "ERROR 53: That file doesn't exist."
  Case 68 : MsgBox "ERROR 68: Drive " & Drive & ": not available."
  Case 76 : MsgBox "ERROR 76: That path doesn't exist."
  Case Else : MsgBox "ERROR " & Err & " occurred."
End Select
Resume Next                                          ' Resume procedure.
```

On...GoSub Statement, On...GoTo Statement

Description Branch to one of several specified lines, depending on the value of an expression.

Syntax 1 **On** *expression* **GoSub** *line-label-list*

Syntax 2 **On** *expression* **GoTo** *line-label-list*

Remarks The argument *expression* can be any numeric expression that evaluates to a value between 0 and 255, inclusive. (The expression is rounded to an integer value before the **On...GoSub** or **On...GoTo** statement is evaluated.) The argument *line-label-list* consists of a list of line numbers and line labels separated by commas. The value of *expression* determines which line the program branches to.

If the value of *expression* is less than 1 or greater than the number of items in the list, one of the following results occurs.

Value	Result
Equal to 0	Program control drops to the next statement.
Greater than number of items in list	Program control drops to the next statement.
Negative	`Illegal function call` error is generated.
Greater than 255	`Illegal function call` error is generated.

You can mix line numbers and line labels in the same list. The number of line labels and/or line numbers you can use with **On...GoTo** and **On...GoSub** is limited only by the number you can physically fit on a single line.

Tip **Select Case** provides more structure and a more flexible way to perform multiple branching.

See Also **GoSub...Return** Statements, **GoTo** Statement, **Select Case** Statement

Example This example uses **On...GoSub** to branch to one of two subroutines depending on user input.

```
N = InputBox("Enter a number greater than 0 and less than 3.")
Msg = "You didn't follow directions."     ' Error message.
On N GoSub Chose1, Chose2                 ' Choose subroutine.
MsgBox Msg : End                          ' Print results.
Chose1:
  Msg = "You chose 1." : Return
Chose2:
  Msg = "You chose 2." : Return
```

On...GoTo Statement

See **On...GoSub** Statement, **On...GoTo** Statement

OnClose Property

See OnOpen, OnClose Properties

OnCurrent Property

Applies To Forms.

Description Specifies the name of a macro or user-defined function to run when the following event occurs: The focus moves from one record to another.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname( )*.

Remarks The event that triggers an OnCurrent macro occurs when you first open a form and whenever the focus leaves one record and moves to another. Microsoft Access runs the macro or function specified by the OnCurrent property before the first or next record is displayed.

You can use this property to run a macro that displays a message or synchronizes records in another form related to the current record. For example, when a customer record becomes current, a macro can display the customer's previous order. When a supplier record is current, the macro can display the products manufactured by the supplier in a Suppliers form.

You can't use the CancelEvent action in an OnCurrent macro. However, you can use the GoToRecord action to move to a different record.

Access Basic

Use a string expression to set the value of this property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

OnDblClick Property

Applies To

Controls (bound object frame, check box,* combo box, command button, list box, option button,* option group, text box, toggle button*) on a form.

*Except when the control is in an option group.

Description

Specifies the name of a macro or user-defined function to run when the following event occurs: You double-click a control or its label in Form view.

Setting

Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname( )*.

Remarks

Microsoft Access runs the macro or function specified by the OnDblClick property when the user double-clicks the control but before the result of the double-click action occurs. You can double-click any selectable control except a subform.

You can use the OnDblClick property to close a dialog box by double-clicking a control instead of choosing the OK button. The macro attached to the OnDblClick property of the control runs a Close action for the dialog box. You could also have the macro hide the dialog box by using the SetValue action to set its Visible property to No.

The result of double-clicking depends on the control. For example, double-clicking a word in a text box selects the entire word. Double-clicking a control containing an OLE object automatically runs the application used to create the object, enabling it to be edited.

You can use a macro attached to the OnDblClick property to modify a control's normal double-click behavior. For example, you can have a macro run before an OLE application is opened in response to double-clicking an embedded object. The macro could display a message box asking if you want to print the object or edit it. You could use a user-defined function to print the object.

You can use the CancelEvent action in an OnDblClick macro to cancel the normal double-click behavior of the control. If you decided to print the object in the preceding example instead of editing it, you would use the user-defined function to print the object and then use a CancelEvent action to cancel opening of the OLE application.

Access Basic

Use a string expression to set the value of this property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

OnDelete Property

See OnInsert, OnDelete Properties

OnEnter, OnExit Properties

Apply To

Controls (bound object frame, check box,* combo box, command button, list box, option button,* option group, subform, text box, toggle button*) on a form.

* Except when the control is in an option group.

Description

Specify the name of a macro or user-defined function to run when one of the following events occurs:

- OnEnter—a control receives the focus.
- OnExit—a control loses the focus.

Setting

Enter the name of the macro you want to run. You can also run a user-defined function by placing an equal sign (=) before the function name and parentheses after it, as in *=functionname*( ).

Remarks

Microsoft Access runs the macro or function specified by the OnEnter property when you go to a control but before the control has the focus. The OnExit property runs the macro or function when you leave a control, but before the control loses the focus.

You can use the OnEnter property to run a macro that displays instructions for the user. For example, you can display a small form or message box showing the type of data the control typically contains. (If the instructions are brief, however, you can display them in the status bar using the StatusBarText property rather than using the OnEnter property and a macro.)

You can use the OnExit or OnEnter property to run a macro that uses a GoToControl action to change the tab order based on current data. For example, if the user enters Male in a Sex field, the focus won't move to a Pregnancy field.

You can use the CancelEvent action in an OnExit macro to cancel exiting the control. You can't use the CancelEvent action in an OnEnter macro. However, you can use the GoToControl action to move the focus to a different control.

Access Basic Use a string expression to set the value of each property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

See Also BeforeUpdate, AfterUpdate Properties; ValidationRule, ValidationText Properties

OnExit Property

See OnEnter, OnExit Properties

OnFormat Property

Applies To Report sections.

Description Specifies the name of a macro or user-defined function to run before the following event occurs: A section is formatted for previewing or printing.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname( )*.

Remarks Microsoft Access runs the macro or function specified by the OnFormat property when it knows which data belongs in the section but before the data is formatted for printing.

If the macro is attached to a detail section, it runs for each record in the section immediately before Microsoft Access formats the data in the record. The macro has access to data in the current record. If the macro is attached to a group header or group footer section, it runs for each new group and has access to data in the group header or group footer. It also has access to the data in the first record in the detail section (if the macro is attached to the group header) or to the data in the last record in the detail section (if the macro is attached to the group footer).

You can use the OnFormat property to run a macro that uses data in the current record to make changes that affect page layout. For example, you can hide or show a congratulatory message next to a salesperson's monthly sales total in a sales report, depending on the sales total. After the macro hides or shows the control, Microsoft Access formats the section using the values of the CanGrow, CanShrink, HideDuplicates, KeepTogether, and Visible property settings.

The OnFormat macro runs for each section of the report. This allows you to make complex running calculations using data from each section, including those not printed.

For changes that don't affect page layout or for macros such as page totals that should run only after the data on a page has been formatted, use the OnPrint property.

You can use the FormatCount property to check whether the OnFormat event has occurred more than once for a record. For example, if a record doesn't fit on the page and part of it moves to the next page, the OnFormat event occurs once for each page, and the FormatCount property is set to 2. You can check the FormatCount value for OnFormat macros that should run only once for a record.

You can use the CancelEvent action in an OnFormat macro to cancel formatting of the section. Microsoft Access doesn't format the section for printing and prints the next section in its place. You can use this action to skip a record in a report without leaving a blank space on the page.

Access Basic Use a string expression to set the value of this property. The expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

See Also FormatCount Property; MoveLayout, NextRecord, PrintSection Properties

---

OnInsert, OnDelete Properties

Apply To Forms.

Description Specify the name of a macro or user-defined function to run when one of the following events occurs:

- OnInsert—the user types the first character in a new record.
- OnDelete—the user deletes a record.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in =*functionname*( ).

Remarks Microsoft Access runs the macro or function specified by the OnInsert property when the user types the first character in a new record but before the record is actually created. The macro or function specified by the OnDelete property runs when the user deletes an existing record but before the record is deleted.

You can use these properties to run macros that display messages or useful information. For example, you can have the OnInsert property specify a macro that displays data the user needs when entering a new record. Similarly, you can use an OnDelete macro to display a message asking whether the user wants to delete a record before it is deleted.

You can use the CancelEvent action in OnInsert and OnDelete macros to cancel inserting and deleting of a record. When you use the CancelEvent action in an OnInsert macro, the focus returns to the new record, which is blank (the character that the user typed is deleted).

Access Basic Use a string expression to set the value of each property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

OnMenu Property

Applies To Forms.

Description Specifies the name of a menu bar macro to attach to the form. This creates a custom menu bar for the form.

Setting Enter the macro name.

Remarks Microsoft Access runs the menu bar macro and displays the custom menu bar when the form is opened. This menu bar appears whenever the form has the focus.

The menu bar macro contains AddMenu actions that create the drop-down menus on the custom menu bar. For each drop-down menu, the menu bar macro names the macro group that defines the commands on the menu. If you make changes to the menu bar macro or the macro groups that define the commands on the drop-down menus while the form is open, you must close the form and reopen it to see the changes in the custom menu bar and its menus.

Menu bar macros can contain only AddMenu actions and are the only macros that can contain AddMenu actions.

You can't use the CancelEvent action (or any action except AddMenu) in an OnMenu macro.

Access Basic

Use a string expression to set the value of this property. You can't use the name of a user-defined function with this property.

Design view: read/write. Other views: read-only.

OnOpen, OnClose Properties

Apply To

Forms. Reports.

Description

Specify the name of a macro or user-defined function to run before or when one of the following events occurs:

- OnOpen—before a form or report is opened.
- OnClose—when a form or report is closed.

Setting

Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname*( ).

Remarks

Microsoft Access runs the macro or function specified by the OnOpen property when the form or report is initially opened but before the first record is displayed. For reports, the macro or function runs before the report prints. The macro or function specified by the OnClose property runs when the form is closed, but before it disappears, or when the report is closed.

You can use the OnOpen property to run a macro that closes another window or presets the focus to move to a particular control on a form. You can also run a macro that asks for information needed before the form or report is opened or printed. For example, you can use the macro to open a pop-up form in which the user enters the criteria for the set of records to display on a form or the date range for a report.

You can use the OnClose property to run a macro that opens another window or requests the user's name to make a log entry indicating who used the form or report.

You can use the CancelEvent action in an OnOpen macro to cancel opening of the form or report. You can use the CancelEvent action in an OnClose macro to cancel closing of the form. However, you can't use the CancelEvent action in an OnClose macro attached to a report.

Notes

- If a form's OnClose property specifies a macro that runs a CancelEvent action, you won't be able to close the form. You must either correct the condition that caused the CancelEvent action or open the macro and delete the CancelEvent action. If the form is a modal form, you won't be able to open the macro.
- If you want to refer to controls from an OnOpen macro, you must first move the focus to the appropriate control or record. For example, to use a SetValue action in an OnOpen macro to set the value of a control on a form, you must first use the GoToControl or GoToRecord action to access the control (or the record containing the control).

Access Basic Use a string expression to set the value of each property. This expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

OnPrint Property

Applies To Report sections.

Description Specifies the name of a macro or user-defined function to run before the following event: You preview or print a section's output.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname*( ).

Remarks Microsoft Access runs the macro or function specified by the OnPrint property after the data in the section is formatted for printing but before the section is printed.

If the macro is attached to a detail section, it runs for each record in the section immediately before Microsoft Access prints the data in the record. The macro has access to data in the current record. If the macro is attached to a group header or group footer section, it runs for each new group and has access to data in the group header or group footer. It also has access to the data in the first record in the detail section (if the macro is attached to the group header) or to the data in the last record in the detail section (if the macro is attached to the group footer).

You can use the OnPrint property with a macro that should be run only when Microsoft Access knows which data will be printed on each page. For example, you can keep running page totals in the page header or footer.

The OnPrint macro runs only for sections that are actually printed.

For changes that affect page layout, such as hiding or showing controls, use the OnFormat property.

You can use the PrintCount property to check whether the OnPrint event has occurred more than once for a record. For example, if part of a record prints on one page and the rest prints on the next page, the OnPrint event occurs twice, and the PrintCount property is set to 2. You can check the PrintCount value for OnPrint macros that should run only once for a record, such as a macro that shows totals for the records on a page in a page footer.

You can use the CancelEvent action in an OnPrint macro to cancel printing of the section. Microsoft Access prints a blank area on the page where the section would have been. You can use this action to skip a record in a report and leave a blank area on the page in its place.

Access Basic Use a string expression to set the value of this property. The expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

See Also MoveLayout, NextRecord, PrintSection Properties; Page Property; PrintCount Property

---

OnPush Property

Applies To Control (command button) on a form.

Description Specifies the name of a macro or user-defined function to run when the following event occurs: The user chooses a command button.

Setting Enter the name of the macro you want to run. To run a user-defined function, place an equal sign (=) before the function name and parentheses after it, as in *=functionname( )*.

Remarks Microsoft Access runs the macro or function specified by the OnPush property when the user chooses (pushes) the command button. The macro or function runs once. If you want the macro or function to run repeatedly while the command button is depressed, set the AutoRepeat property to Yes.

You use the OnPush property to run the macros that contain the actions for each command button on a form. For example, by choosing one command button on a form, you can print a report showing the form's records. By choosing another, you can open a related form.

You can't use the CancelEvent action in an OnPush macro.

Access Basic

Use a string expression to set the value of this property. The expression can be the name of a macro or user-defined function. The function name must be preceded by an equal sign and followed by parentheses.

Design view: read/write. Other views: read-only.

Open Statement

Description

Enables input/output (I/O) to a file.

Syntax

Open *file* [**For** *mode*] [**Access** *access*] [*lock*] **As** [#]*filenumber* [**Len** = *reclen*]

Remarks

You must open a file before any I/O operation can be performed on it. **Open** allocates a buffer for I/O to the file and determines the mode of access used with the buffer.

The **Open** statement uses these arguments.

Argument	Description
file	File name or path.
mode	Reserved word that specifies the file mode: **Append**, **Binary**, **Input**, **Output**, **Random**.
access	Reserved word that specifies which operations are permitted on the open file: **Read**, **Write**, **Read Write.**
lock	Reserved word that specifies which operations are permitted on the open file by other processes: **Shared**, **Lock Read**, **Lock Write**, **Lock Read Write**.
filenumber	Integer expression with a value between 1 and 255, inclusive. When an **Open** statement is executed, *filenumber* is associated with the file as long as it is open. Other I/O statements can use the number to refer to the file.
reclen	For files opened for random access, the record length; for sequential files, the number of characters buffered. The argument *reclen* is a positive integer expression less than or equal to 32,767 bytes.

If *file* doesn't exist, it is created when a file is opened for **Output**, **Random**, **Binary**, or **Append** modes.

The argument *mode* is a reserved word that specifies one of the following file modes.

Mode	Description
Random	Random-access file mode (default). In this mode, if no **Access** clause is present, three attempts are made to open the file when the **Open** statement is executed. Access is attempted in the following order: 1. Read and write 2. Write only 3. Read only
Binary	Binary file mode. In **Binary** mode, you can read and write information to any byte position in the file using **Get** and **Put** statements. If no **Access** clause is present, three attempts are made to open the file, following the same order as for **Random** files.
Input	Sequential input mode.
Output	Sequential output mode.
Append	Sequential output mode. **Append** sets the file pointer to the end of the file. A **Print #** or **Write #** statement then extends (appends to) the file.

If the argument *mode* is omitted, the default mode (**Random**) is used.

The argument *access* is a reserved word that specifies the operations that can be performed on the opened file. If the file is already opened by another process and the specified type of access is not allowed, the **Open** operation fails and a `Permission denied` error occurs.

The **Access** clause works only if you are using a version of MS-DOS that supports networking (MS-DOS version 3.1 or later). You (or the network startup program) must also run the SHARE.EXE program to enable locking operations. If you use the **Access** clause with a version of MS-DOS that doesn't support networking, a `Feature unavailable` error occurs.

The argument *access* can be one of the following reserved words.

Access type	Description
Read	Opens the file for reading only.
Write	Opens the file for writing only.
Read Write	Opens the file for both reading and writing. This mode is valid only for **Random** and **Binary** files and files opened for **Append** mode.

The *lock* clause works in a multiuser or multiprocessing environment to restrict access by other processes or users to an open file. The argument can be one of the following reserved words.

Lock type	Description
Shared	Any process on any machine can read from or write to this file.
Lock Read	No other process is granted read access to this file. This lock is applied only if no other process has a previous read access to the file.
Lock Write	No other process is granted write access to this file. This lock is applied only if no other process has a previous write access to the file.

If you don't specify a lock type, the file can be opened for reading and writing any number of times by the statement, but other operations are denied access to the file while it is open.

When **Open** is restricted by a previous process, a `Permission denied` error occurs.

The argument *reclen* is an integer expression less than or equal to 32,767 bytes. It specifies settings for **Random** and sequential files.

For **Random** files, the argument *reclen* sets the record length (the number of characters in a record). The default record length is 128 bytes.

For sequential files, the argument *reclen* specifies the number of characters to be loaded into the buffer before the buffer is written to or read from the disk. A larger buffer uses more memory but provides faster file I/O. Conversely, a smaller buffer leaves more room in memory but slows I/O. The default buffer size is 512 bytes.

The **Len** clause and *reclen* are ignored if *mode* is **Binary**. For sequential files, *reclen* need not correspond to an individual record size, because a sequential file may have records of different sizes.

Note In **Input**, **Random**, and **Binary** modes, you can open a file using a different file number without first closing the file. In **Output** and **Append** modes, you must close a file before opening it with a different file number.

See Also **Close** Statement, **FreeFile** Function

Example This example uses **Open** to open a file for output.

```
FileName = "TESTFILE"
Open FileName For Output As FreeFile          ' Open file.
```

OpenDatabase Function

Description Opens a specified database and returns a database object for that database.

Syntax **OpenDatabase**(*database* [, *exclusive* [, *read-only*]])

Remarks When you want to use Access Basic code to directly access your data, you must use a database. Usually, the current database is the one you want to use; the **CurrentDB** function provides the necessary access. If you have only one database open at a time, the open database is the current database.

However, if you need to open a specific database (which may not be the current database), use the **OpenDatabase** function instead.

This function can be used only in Access Basic code.

The **OpenDatabase** function uses the following arguments.

Argument	Description
database	A string expression that is the name of an existing database.
exclusive	A Boolean value that is **True** (–1) if the database is to be opened for exclusive (nonshared) access and **False** (0) if the database is to be opened for shared access. If this argument is omitted, the database is opened for shared access.
read-only	A Boolean value that is **True** if the database is to be opened for read-only access and **False** if it is to be opened for read/write access. If this argument is omitted, the database is opened for read/write access.

The *database* argument can include the fully qualified DOS path. If you omit the path, only the current directory is searched. If *database* isn't on a local drive, you must specify a fully qualified network path. The following example shows how you might specify a database in a network environment.

```
"\\MYSERVER\MYSHARE\MYDIR\MYDB"
```

If *database* refers to a database that is already open for exclusive access by another user, an error occurs when you use **OpenDatabase**. If *database* refers to the current database, use the **CurrentDB** function instead.

If *database* doesn't refer to an existing database, an error occurs.

See Also **Close** Method, **CurrentDB** Function

Example

This example opens the Employees table in the NWIND.MDB database. If the database can't be opened (because, for example, it has already been opened for exclusive access by another user in a multiuser environment), an error occurs and a message appears.

```
Sub Open_Northwind ()
  Const MB_ICONEXCLAMATION = 48
  Dim MyDB As Database, MyTable As Table, MyFile As String
  Dim ErrorCondition As Integer
  MyFile = "NWIND.MDB"                                  ' Define name of database.
  On Error GoTo DB_Error_Handler                        ' Enable error trapping.
  Set MyDB = OpenDatabase(MyFile)                       ' Open database.
  If Not ErrorCondition Then
    On Error GoTo Table_Error_Handler                   ' Enable error trapping.
    Set MyTable = MyDB.OpenTable("Employees")           ' Open Table.
    If Not ErrorCondition Then
      On Error GoTo Edit_Error_Handler                  ' Enable error trapping.
      Do Until MyTable.EOF
        If MyTable!Title = "Sales Representative" Then
          MyTable.Edit                                  ' Enable changes.
          MyTable!Title = "Account Executive"           ' Make changes.
          MyTable.Update                                ' Save changes.
        End If
          MyTable.MoveNext                              ' Move to next record.
      Loop
      MyTable.Close                                     ' Close Table.
    End If
    MyDB.Close                                          ' Close database.
  End If
  On Error GoTo 0                                       ' Disable error trapping.
  Exit Sub

DB_Error_Handler:
  ErrorCondition = True
  MsgBox "Cannot open NWIND.MDB database.", MB_ICONEXCLAMATION
  Resume Next

Table_Error_Handler:
  ErrorCondition = True
  MsgBox "Cannot open Employees table.", MB_ICONEXCLAMATION
  Resume Next
```

```
Edit_Error_Handler:
  ErrorCondition = True
  MsgBox "Cannot edit Employees table.", MB_ICONEXCLAMATION
  Resume Next

End Sub
```

Tip Using an update query to change job titles might be more efficient.

OpenForm Action

Description Opens a form in Form view, Design view, Print Preview, or Datasheet view. You can select data entry and window modes for the form and restrict the records the form displays.

Action argument	Description
Form Name	The name of the form to open. This argument's drop-down list in the Action Arguments section of the Macro window shows all forms in the current database. Required argument.
View	The view in which to open the form. Select Form, Design, Print Preview, or Datasheet from the drop-down list. The default is Form.
Filter Name	A filter that restricts or sorts the form's records. You can enter the name of either an existing query or a filter that was saved as a query. However, the query must include all the fields in the form you are opening, or it must have been saved with the Restrict Available Fields option in the Query Properties dialog box cleared.
Where Condition	A valid SQL WHERE clause or expression that Microsoft Access uses to select records from the form's underlying table or query. If you select a filter with the Filter Name argument, Microsoft Access applies this WHERE clause to the filter. To open a form in which the value of one of its controls equals the value of a control on another form, use this syntax: [*controlname on opened form*] = ➥ **Forms**![*formname*]![*controlname on other form*]

Action argument	Description
	In contrast to other action arguments, if you use only the name of a control in the expression for this argument, it refers to a control on the form being opened, not the form that called the macro.
Data Mode	The data entry mode for the form. This applies only to forms opened in Form view or Datasheet view. Select Add (the user can add new records but can't edit existing records), Edit (the user can edit existing records and add new records), or Read Only (the user can only view records). The default is Edit.
Window Mode	The window mode in which the form opens. Select Normal (the form is in the mode set by its properties), Hidden (the form is hidden), Icon (the form opens as an icon at the bottom of the screen), or Dialog (the form's Modal and PopUp properties are set to Yes). The default is Normal.

Remarks

This action is similar to choosing the Open button or Design button in the Database window after clicking the Form button and selecting a form. With this action you can select additional options.

A form can be modal (it must be closed or hidden before the user can perform any other action) or modeless (the user can move to other windows while the form is open). It can also be a pop-up form (a fixed-size, movable form used to collect or display information). You set the Modal and PopUp properties when you design the form. If you select Normal for the Window Mode argument, the form opens in the mode specified by these property settings. If you select Dialog for the Window Mode argument, these properties are both set to Yes. A form opened as Hidden or as an Icon returns to the mode specified by its property settings when you show or restore it.

When you open a form with the Window Mode argument set to Dialog, Microsoft Access suspends the macro until the form is closed or hidden. You can hide a form by setting its Visible property to No using the SetValue action.

Switching to Design view while the form is open removes most of the argument settings for the form, such as the Data Mode and Window Mode settings. They aren't in effect even if the user returns to Form view or Datasheet view.

See Also

Close Action, SelectObject Action, SetValue Action

Access Basic

Syntax

DoCmd OpenForm *formname* [, *view*] [, *filtername*] [, *wherecondition*] [, *datamode*] ➥ [, *windowmode*]

Argument	Description
formname	A string expression that is the valid name of a form in the current database.
view	One of the following intrinsic constants: A_NORMAL A_DESIGN A_PREVIEW A_FORMDS A_NORMAL opens the form in Form view. If you leave this argument blank, the default (A_NORMAL) is assumed.
filtername	A string expression that is the valid name of a query in the current database.
wherecondition	A string expression that is a valid SQL WHERE clause.
datamode	One of the following intrinsic constants: A_ADD A_EDIT A_READONLY If you leave this argument blank, the default (A_EDIT) is assumed.
windowmode	One of the following intrinsic constants: A_NORMAL A_HIDDEN A_ICON A_DIALOG If you leave this argument blank, the default (A_NORMAL) is assumed.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example opens `My Form` in Form view and displays only records with `Jackson` in the `Last Name` field. The displayed records can be edited, and new records can be added.

```
DoCmd OpenForm "My Form", , ,"[Last Name] = ""Jackson"""
```

OpenQuery Action

Description Opens a select or crosstab query in Datasheet view, Design view, or Print Preview. Runs an action query. You can also select a data entry mode for the query.

Action argument	Description
Query Name	The name of the query to open. This argument's drop-down list in the Action Arguments section of the Macro window shows all queries in the current database. Required argument.
View	The view in which to open the query. Select Datasheet, Design, or Print Preview from the drop-down list. The default is Datasheet.
Data Mode	The data entry mode for the query. This applies only to queries opened in Datasheet view. Select Add (the user can add new records but can't edit existing records), Edit (the user can edit existing records and add new records), or Read Only (the user can only view records). The default is Edit.

Remarks If you select Datasheet for the View argument, Microsoft Access displays the dynaset if the query is a select or crosstab query and runs the query if it's an action query.

The OpenQuery action is similar to choosing the Open button or Design button in the Database window after clicking the Query button and selecting a query. With this action you can select additional options.

Switching to Design view while the query is open removes the Data Mode argument setting for the query. This setting isn't in effect even if the user returns to Datasheet view.

Tip If you don't want to display the system messages that normally appear when an action query is run (indicating it is an action query and showing how many records will be affected), you can use the SetWarnings action to suppress the display of these messages.

See Also SetWarnings Action

Access Basic

Syntax

DoCmd OpenQuery *queryname* [, *view*] [, *datamode*]

Argument	Description
queryname	A string expression that is the valid name of a query in the current database.

view	One of the following intrinsic constants:
	`A_NORMAL`
	`A_DESIGN`
	`A_PREVIEW`
	If *queryname* is the name of a select query, `A_NORMAL` opens the query's dynaset. If *queryname* refers to an action query, `A_NORMAL` runs the query.
	If you leave this argument blank, the default (`A_NORMAL`) is assumed.

datamode	One of the following intrinsic constants:
	`A_ADD`
	`A_EDIT`
	`A_READONLY`
	If you leave this argument blank, the default (`A_EDIT`) is assumed.

Remarks

If you specify the *datamode* argument and leave the *view* argument blank, you must include the *view* argument's comma. If you leave a trailing argument blank, don't use a comma following the last argument you specify.

Example

This example opens `My Query` in Normal view and allows records to be viewed but not edited or added.

```
DoCmd OpenQuery "My Query", , A_READONLY
```

OpenQueryDef Method

Description Opens a specified **QueryDef** for subsequent editing.

Syntax **Set** *querydef* = *database*.**OpenQueryDef**(*name*)

Remarks The **OpenQueryDef** method uses the following arguments.

Argument	Description
querydef	Name of the **QueryDef** you want to use during editing
database	Name of an open database object
name	String expression that is the name of an existing **QueryDef**

To reuse a **QueryDef** name, you must do one of the following: recreate it using the **DeleteQueryDef** and **CreateQueryDef** methods; change the SQL, Sort, and Filter properties (you must first open the **QueryDef**); or choose Rename from the File menu.

If the **QueryDef** contains parameters, you must provide the corresponding arguments. The parameter settings are dependent upon the data type of the parameters. An **Integer** parameter requires an argument that is also an **Integer**, a **Double** requires a **Double**, and so on. If a parameter remains unset at run time, an error occurs.

The following example sets a parameter named `FindAmount`.

```
MyQuery!FindAmount = 1000
```

You can use the **ListParameters** method to create a **Snapshot** that lists all of the parameters in the **QueryDef**.

You can use the **CreateDynaset** and **CreateSnapshot** methods on a **QueryDef** even if it is open.

If *name* refers to a **QueryDef** that doesn't exist, an error occurs.

See Also **Close** Method, **CreateQueryDef** Method, **DeleteQueryDef** Method, **ListParameters** Method

Example This example shows how the **OpenQueryDef** method works using the Orders table in the NWIND.MDB database. `Create_Large_Orders` creates a new **QueryDef** that contains a Currency parameter. `Open_Large_Orders` provides a value for the parameter and creates a **Snapshot** that contains the records for all orders greater than 1000.

```
Sub Create_Large_Orders
  Dim MyDB As Database, MyQuery As QueryDef, ListSet As Snapshot
  Set MyDB = CurrentDB()
  Set MyQuery = MyDB.CreateQueryDef("Large Orders")        ' Create query.
  MyQuery.SQL = "Parameters FindAmount Currency; SELECT * FROM Orders
  ➥ WHERE [Order Amount] > FindAmount;
End Sub
```

```
Sub Open_Large_Orders
  Dim MyDB As Database, MyQuery As QueryDef, ListSet As Snapshot
  Set MyDB = CurrentDB()
  Set MyQuery = MyDB.OpenQueryDef("Large Orders")          ' Open QueryDef.
  MyQuery!FindAmount = 1000                                ' Set parameter.
  Set ListSet = MyQuery.CreateSnapshot("Large Orders")     ' Create Snapshot.
  MyQuery.Close                                            ' Close QueryDef.
  MyDB.DeleteQueryDef("Large Orders")                      ' Delete QueryDef.
End Sub
```

OpenReport Action

Description Opens a report in Design view or Print Preview or prints the report immediately. You can also restrict the records that are printed in the report.

Action argument	Description
Report Name	The name of the report to open. This argument's drop-down list in the Action Arguments section of the Macro window shows all reports in the current database. Required argument.
View	The view in which to open the report. Select Print (to print the report immediately), Design, or Print Preview from the drop-down list. The default is Print Preview.
Filter Name	A filter that restricts or sorts the report's records. You can enter the name of either an existing query or a filter that was saved as a query. However, the query must include all the fields in the report you are opening, or it must have been saved with the Restrict Available Fields option in the Query Properties dialog box cleared.

Action argument	Description
Where Condition	A valid SQL **WHERE** clause or expression that Microsoft Access uses to select records from the report's underlying table or query. If you select a filter with the Filter Name argument, Microsoft Access applies this WHERE clause to the filter. To open a report in which the value of one of its controls equals the value of a control on a form, use this syntax: [*controlname on opened report*] = ➥ **Forms**![*formname*]![*controlname on form*] In contrast to other action arguments, if you use only the name of a control in the expression for this argument, it refers to a control on the report being opened, not the form or report that called the macro.

Remarks

The Print setting for the View argument prints the report immediately using the latest printer settings without bringing up the Print dialog box. You can also use the OpenReport action to open and set up a report, and then use the Print action to print it. For example, you may want to modify the report or use the Print action to change the printer settings before you print.

The OpenReport action is similar to choosing the Design button, the Preview button, or the Print command from the File menu in the Database window after clicking the Report button and selecting a report. With this action you can select additional options.

Tips

- To print similar reports for different sets of data, use a filter or a **WHERE** clause to restrict the records printed in the report. Then edit the macro to apply a different filter or change the Where Condition argument.
- You can select a report in the Database window and drag it to a macro action row. This automatically creates an OpenReport action that opens the report in Print Preview mode.

Access Basic

Syntax

DoCmd OpenReport *reportname* [, *view*] [, *filtername*] [, *wherecondition*]

Argument	Description
reportname	A string expression that is the valid name of a report in the current database.

view	One of the following intrinsic constants:
	`A_NORMAL`
	`A_DESIGN`
	`A_PREVIEW`
	`A_NORMAL` prints the report immediately.
	If you leave this argument blank, the default (`A_PREVIEW`) is assumed.

filtername	A string expression that is the valid name of a query in the current database.
wherecondition	A string expression that is a valid SQL WHERE clause.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example prints `My Report` using the existing query `Report Filter`.

```
DoCmd OpenReport "My Report", A_NORMAL, "Report Filter"
```

OpenTable Action

Description

Opens a table in Datasheet view, Design view, or Print Preview. You can also select a data entry mode for the table.

Action argument	Description
Table Name	The name of the table to open. This argument's drop-down list in the Action Arguments section of the Macro window shows all tables in the current database. Required argument.
View	The view in which to open the table. Select Datasheet, Design, or Print Preview from the drop-down list. The default is Datasheet.
Data Mode	The data entry mode for the table. This applies only to tables opened in Datasheet view. Select Add (the user can add new records but can't edit existing records), Edit (the user can edit existing records and add new records), or Read Only (the user can only view records). The default is Edit.

Remarks

This action is similar to choosing the Open button or Design button in the Database window after clicking the Table button and selecting a table. With this action you can select additional options.

Switching to Design view while the table is open removes the table's Data Mode argument setting. This setting isn't in effect even if the user returns to Datasheet view.

Access Basic

Syntax

DoCmd OpenTable *tablename* [, *view*] [, *datamode*]

Argument	Description
tablename	A string expression that is the valid name of a table in the current database.
view	One of the following intrinsic constants: A_NORMAL A_DESIGN A_PREVIEW A_NORMAL opens the table in Datasheet view. If you leave this argument blank, the default (A_NORMAL) is assumed.

Argument	Description

datamode	One of the following intrinsic constants:
	`A_ADD`
	`A_EDIT`
	`A_READONLY`
	If you leave this argument blank, the default (`A_EDIT`) is assumed.

Remarks

If you specify the *datamode* argument and leave the *view* argument blank, you must include the *view* argument's comma. If you leave a trailing argument blank, don't use a comma following the last argument you specify.

Example

This example opens `My Table` in Print Preview.

```
DoCmd OpenTable "My Table", A_PREVIEW
```

OpenTable Method

Description Creates a **Table** object from a specified table for subsequent editing.

Syntax **Set** *table* = *database*.**OpenTable**(*name* [, *exclusive*])

Remarks The **OpenTable** method uses the following arguments.

Argument	Description
table	Name of the table you want to use during editing.
database	Name of an open database.
name	String expression that is the name of an existing table in *database*. It cannot be the name of a linked table; for linked tables, use the **CreateDynaset** and **CreateSnapshot** methods.
exclusive	Boolean value that is **True** (–1) if *table* is to be opened for exclusive (nonshared) access and **False** (0) if *table* is to be opened for shared access. If this argument is omitted, shared access is enabled.

If you are interested in just part of a table, you can use the **CreateDynaset** method to create a subset of the data. Although fields in the table referred to by *name* can be ordered

only according to its indexes, you can sort a **Dynaset** on any field. Depending on the selection criteria you use to create it, a **Dynaset** will generally contain fewer records.

You can also use the **CreateSnapshot** method instead of opening the table. However, data in a **Snapshot** can't be changed.

If the table is opened for exclusive access when you use **OpenTable**, an error occurs.

See Also **Close** Method, **ListFields** Method, **ListIndexes** Method, **ListTables** Method

Example This example opens the Customers table in the NWIND.MDB database and creates a **Snapshot** that contains all of the table's indexes.

```
Dim MyDB As Database, ListSet As Snapshot, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Customers")       ' Open Table.
Set ListSet = MyTable.ListIndexes()             ' Create Snapshot.
...
MyTable.Close                                   ' Close Table.
```

Operator Precedence

Description When several operations occur in an expression, each part is evaluated and resolved in a predetermined order. That order is known as operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Within individual categories, operators are evaluated in the order of precedence shown below.

Arithmetic	Comparison	Logical
Exponentiation (^)	*Equality (=)	**Not**
Negation (–)	*Inequality (< >)	**And**
Multiplication and division (*, /)	*Less than (<)	**Or**
Integer division	*Greater than (>)	**Xor**
Modulo arithmetic (**Mod**)	*Less than or equal to (<=)	**Eqv**
Addition and subtraction (+, –)	*Greater than or equal to (>=)	**Imp**
String concatenation (&)	***Like**	

* All comparison operators have equal precedence and are evaluated in the left-to-right order in which they appear.

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation operator (**&**) is not really an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. Similarly, the **Like** operator, while equal in precedence to all comparison operators, is actually a pattern-matching operator.

See Also & Operator, * Operator, + Operator, – Operator, / Operator, \ Operator, ^ Operator, **And** Operator, Comparison Operators, **Eqv** Operator, **Imp** Operator, **Like** Operator, **Mod** Operator, **Not** Operator, **Or** Operator, **Xor** Operator

Option Base Statement

Description Declares the default lower bound for array subscripts.

Syntax **Option Base** *number*

Remarks The **Option Base** statement is never required. However, if used, it can appear only once in a module and can occur only in the Declarations section. If you choose to use an **Option Base** statement, it must be used before you declare the dimensions of any arrays.

The value of *number* must be either 0 or 1. The default base is 0.

Tip The **To** clause in the **Dim**, **Global**, **ReDim**, and **Static** statements provides a more flexible way to control the range of an array's subscripts. However, if you don't explicitly set the lower bound with a **To** clause, you can use **Option Base** to change the default lower bound to 1.

See Also **Dim** Statement, **Global** Statement, **LBound** Function, **ReDim** Statement, **Static** Statement

Example This example uses the **Option Base** statement to override the default base array subscript value of 0.

```
Option Base 1                          ' Goes in module Declarations section.
Dim X(10) As Single                    ' X has 10 elements, from 1 through 10.
```

Option Compare Statement

Description Declares the default comparison mode to use when text data is compared.

Syntax **Option Compare** {**Binary** | **Database** | **Text**}

Remarks The **Option Compare** statement is used in the Declarations section of a module to specify the text comparison method for that module.

Text comparisons are **Binary, Database,** or **Text**.

Binary comparisons are case-sensitive; so, for example, an uppercase M doesn't match a lowercase M. When determining whether a character is less than, equal to, or greater than another character, the program uses the character's relative order of appearance in the ANSI character set.

Text comparisons are not case-sensitive; so, for example, an uppercase M matches a lowercase M. In addition, the relative order of characters is not the same as it is for **Binary**. For **Text** comparisons, the relative order of characters is determined by the setting of the country code in the International section of your WIN.INI file.

Database comparisons are based on the characters' relative position according to the sort order specified when the database was created or compacted.

If the module doesn't include an **Option Compare** statement, the default is **Binary**.

See Also Comparison Operators, **InStr** Function, **StrComp** Function

Example Use this statement in the Declarations section of any module to specify the **Database** comparison mode for that module. For Microsoft Access, this statement is automatically placed in each new module you create.

```
Option Compare Database
```

Option Explicit Statement

Description Used to force explicit declaration of all variables.

Syntax **Option Explicit**

Remarks The **Option Explicit** statement is used in the Declarations section of a module to force explicit declaration of all variables for that module.

If you don't use the **Option Explicit** statement, all undeclared variables are of **Variant** data type (unless the default type is otherwise specified with a **Def***type* statement). When you use the **Option Explicit** statement, all variables must be explicitly declared before they are used. If you attempt to use an undeclared variable name, an error occurs.

Tip Use **Option Explicit** to avoid mistyping the name of an existing variable or risking confusion in code in which the scope of the variable is not clear.

See Also **Const** Statement, **Dim** Statement, **Function** Statement, **Global** Statement, **Static** Statement, **Sub** Statement

Example Use this statement in the Declarations section of a module to specify that no variables can be implicitly created.

```
Option Explicit
```

OptionValue Property

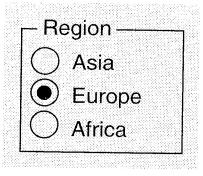
Applies To Controls (check box, option button, toggle button) in an option group.

Description Sets a numerical value for a control. When the control is selected, the number is assigned to the option group.

If the option group is bound to a field, this value is also stored in the field. The number in the field determines which control in the option group is selected when a record becomes current.

Remarks Each control in an option group has a numeric value that you can set with the OptionValue property.

In this example, the option group is bound to the Region field in a table. The Asia button has been assigned an OptionValue setting of 1, the Europe button has been set to 2, and the Africa button has been set to 3. When a button on the form is selected, the Region field will be set to the same number as the button. In this case, because the setting of the Europe button is 2, Region also equals 2.



Unless you set the OptionValue property yourself, the first control you draw in an option group assumes a value of 1, the second control assumes a value of 2, and so on.

Note When a toggle button, an option button, or a check box is not in an option group, the control has no OptionValue property. Instead, each such control has a ControlSource property, and the value of each control is either **True** (–1) or **False** (0).

Access Basic

The OptionValue property setting is a **Long** value. Use a numeric expression to set the value of this property.

Design view: read/write. Other views: read-only.

Or Operator

Description

Used to perform a logical disjunction on two expressions.

Syntax

result = *expr1* **Or** *expr2*

Remarks

If either or both expressions evaluate true (nonzero), *result* is **True** (–1). The following table illustrates how *result* is determined.

If *expr1* is	And *expr2* is	*result* is
true (nonzero)	true	**True** (–1)
true	false (0)	**True**
true	**Null**	**True**
false	true	**True**
false	false	**False** (0)
false	**Null**	**Null**
Null	true	**True**
Null	false	**Null**
Null	**Null**	**Null**

The **Or** operator also performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

If bit in *expr1* is	And bit in *expr2* is	*result* is
0	0	0
0	1	1
1	0	1
1	1	1

Bit-wise comparisons can be performed only in Access Basic code.

See Also

And Operator, **Eqv** Operator, **Imp** Operator, **Not** Operator, Operator Precedence, **Xor** Operator

Example This example prints a message that depends on the value of variables `A`, `B`, and `C`, assuming that no variable is a **Null**. If `A` = 10, `B` = 8, and `C` = 11, the left expression is **True** and the right expression is **False**. Because at least one comparison expression is **True**, the **Or** expression evaluates **True**.

```
If A > B Or B > C Then
  Debug.Print "One or both comparison expressions are True."
Else
  Debug.Print "Both comparison expressions are False."
End If
```

Page Property

Applies To Reports.

Description Specifies the current page number when a report is printed.

Setting You can use the Page property only in an expression, a macro, or Access Basic.

Remarks You can use the Page property in an expression ("=Page") set by the ControlSource property of a text box in a page header or page footer. You can also use this property to put page numbers on printed reports. Use it in an expression to reset the page number for each group in a report.

Access Basic The value of this property is an integer.

Design view: not available. Other views: read/write.

See Also PageHeader, PageFooter Properties; SetValue Action

PageFooter Property

See PageHeader, PageFooter Properties

PageHeader, PageFooter Properties

Apply To

Reports.

Description

Specify whether page headers and page footers are printed on the same pages as report headers and report footers.

Setting

The PageHeader and PageFooter property settings are:

Setting	Description
All Pages	(Default) Page footer or header section prints on all pages of a report.
Not with Rpt Hdr	Page footer or header section does not print on the same page as a report header.
Not with Rpt Ftr	Page footer or header section does not print on the same page as a report footer.
Not with Rpt Hdr/Ftr	Page footer or header section does not print on a page that has either a report header or report footer.

Remarks

Printed reports can have information printed at the top (headers) or at the bottom (footers) of the page. For example, your company name can appear at the top and the date and page number at the bottom of each page in a report.

In a report's Design view, use the Page Hdr/Ftr command from the Layout menu to display the page header and page footer sections.

Note When forms are printed, page headers and page footers always print on all pages.

Access Basic

The PageHeader and PageFooter property settings and their values are:

Setting	Value
All Pages	0
Not with Rpt Hdr	1
Not with Rpt Ftr	2
Not with Rpt Hdr/Ftr	3

Design view: read/write. Other views: read-only.

Parent Property

Applies To Control (subform/subreport).

Description Refers to the parent (main) form or report into which a subform or subreport is inserted.

Setting The Parent property has no settings.

Usage Design view: read-only. Other views: read-only.

Remarks You can use the Parent property when you have a subform or subreport that has been inserted in multiple main forms or reports.

For instance, you might insert an Order Detail subform into both a form and a report. The following example uses the Parent property to refer to the `Order ID` field, which is present on the main form and report. You can enter this expression in a bound control on the subform.

```
= Parent![Order ID]
```

If you use the Parent property with a control that has no parent, an error occurs.

See Also **Forms** Object, SourceObject Property

Partition Function

Description Returns a string indicating where a number occurs within a calculated series of ranges.

Syntax **Partition**(*number*, *start*, *stop*, *interval*)

Remarks The **Partition** function is most useful in queries. You can create a select query that shows how many orders fall in various ranges (for example, order values from 1 to 1000, 1001 to 2000, and so on).

You might also use **Partition** to show whether you are getting orders out on time. You can design a crosstab query that shows how many orders arrive on time, 1 to 5 days late, 6 to 10 days late, and so on. You can even create a graph that summarizes this information.

The **Partition** function uses the following arguments.

Argument	Description
number	**Long** number that you want to evaluate against the ranges.
start	**Long** number that is the start of the overall range of numbers. It can't be less than 0.
stop	**Long** number that is the end of the overall range of numbers. It can't be equal to or less than *start*.
interval	**Long** number that is the interval spanned by each range in the series from *start* to *stop*. It can't be less than 1.

The following table shows how the ranges are determined using three sets of *start*, *stop*, and *interval* arguments. The First range and Last range columns illustrate the text **Partition** returns. The ranges are represented by *lowervalue:uppervalue*, where the low end (*lowervalue*) of the range is separated from the high end (*uppervalue*) with a colon (:).

start	*stop*	*interval*	Before first	First range	Last range	After last
0	99	5	" : -1"	" 0: 4"	" 95: 99"	" 100: "
20	199	10	" : 19"	" 20: 29"	" 190: 199"	" 200: "
100	1010	20	" : 99"	" 100: 119"	" 1000: 1010"	" 1011: "

In this table, the third example shows the result when *start* and *stop* define a set of numbers that can't be evenly divided by *interval*. The last range extends to *stop* (11 numbers) even though *interval* is 20.

The **Partition** function identifies the particular range in which *number* falls and returns a string that describes that range. For example, if you specify a range of values from 1 to 100 and an interval of 5, the following ranges result when *number* is above, below, or within the range of numbers.

Number	Value is	Range
111	Above	" 101: "
0	Below	" : 0"
44	Within	" 41: 45"

If necessary, **Partition** returns a range with enough leading spaces so that there are the same number of characters to the left and right of the colon as there are characters in the *stop* argument, plus one. This ensures that if you use **Partition** with other numbers, the resulting text will be handled properly during any subsequent sort operation.

If *interval* is 1, the range is *number:number*, regardless of the *start* and *stop* arguments. For example, if *interval* is 1, *number* is 100 and *stop* is 1000, **Partition** returns " 100: 100".

If any of the arguments is **Null**, **Partition** returns a **Null**.

Example

The first example creates a select query that inspects the `Order Amount` field in the Orders table of the NWIND.MDB database. It uses the **Partition** and **Count** functions to evaluate and count the number of orders in each of the ranges. The ranges are defined by the arguments to **Partition**: *start* = `1`, *stop* = `20000`, *interval* = `1000`.

You can enter the expression below in the SQL dialog box.

```
SELECT DISTINCTROW Partition([Order Amount], 1, 20000, 1000) AS Range,
➥ Count(Orders.[Order Amount]) AS [Count] FROM Orders
➥ GROUP BY Partition([Order Amount], 1, 20000, 1000)
➥ ORDER BY Partition([Order Amount], 1, 20000, 1000),
➥ Count(Orders.[Order Amount]);
```

The result of the query could look like the following table.

Range	Count
1: 1000	588
1001: 2000	274
2001: 3000	111
3001: 4000	38
4001: 5000	33
5001: 6000	6
6001: 7000	8
7001: 8000	2
8001: 9000	4
9001: 10000	3
10001: 11000	6
11001: 12000	2
12001: 13000	1
13001: 14000	1
16001: 17000	1

This table contains no orders for the ranges `14001:15000` and `15001:16000`.

The next example creates a select query that inspects the `Order Amount` field in the Orders table of the NWIND.MDB database. For each customer, it calculates the number of orders that fall within each of several ranges. The ranges are defined by the arguments to **Partition**: *start* = `1`, *stop* = `20000`, *interval* = `1000`.

You can enter the expression below in the SQL dialog box.

```
SELECT Orders.[Customer ID],
➥ Partition(Int([Order Amount]), 1, 20000, 1000) AS Range,
➥ Count(Orders.[Order ID]) AS [Count]
➥ FROM Orders
➥ GROUP BY Orders.[Customer ID],
➥ Partition(Int([Order Amount]), 1, 20000, 1000);
```

The result of the select query could look like the following table.

Customer ID	Range	Count
ALWAO	1: 1000	7
ANDRC	1: 1000	4
ANTHB	1: 1000	10
ANTHB	1001: 2000	2
ANTHB	2001: 3000	1
AROUT	1: 1000	9
AROUT	1001: 2000	3
AROUT	2001: 3000	1
AROUT	4001: 5000	1
BABUJ	1: 1000	10
BABUJ	1001: 2000	7
BABUJ	2001: 3000	2
BERGS	1: 1000	8
:	:	:

In this table, customer AROUT has 14 orders. Nine are valued between 1 and 1000, three are valued between 1001 and 2000, and so on.

The following example creates a crosstab query that inspects the Order Amount field in the Orders table of the NWIND.MDB database. It calculates the number of orders for each customer that fall within each of several ranges. The ranges are defined by the arguments to **Partition**: *start* = 1, *stop* = 20000, *interval* = 1000.

You can enter the expression below in the SQL dialog box.

```
TRANSFORM Count(Orders.[Order ID]) AS [CountOfOrder ID]
➥ SELECT Orders.[Customer ID] FROM Orders GROUP BY Orders.[Customer ID]
➥ PIVOT Partition(Int([Order Amount]), 1, 20000, 1000);
```

The result of the crosstab query could look like the following table.

Customer ID	1: 1000	1001: 2000	2001: 3000	3001: 4000
ALWAO	7			
ANDRC	4			
ANTHB	10	2	1	
AROUT	9	3	1	1
BABUJ	10	7	2	
BERGS	8	3		
BLUEL	2	5	1	
BLUMG	1		1	
:				

Picture Property

Applies To Controls (command button, toggle button).

Description Determines whether a bitmap is loaded into a command button or toggle button.

Setting Enter the path and file name of the bitmap. The bitmap is loaded into and stored with the button.

The default setting is "(none)." After you have entered the file name and the bitmap is loaded into the button, the property setting displays "(bitmap)." If you delete "(bitmap)" from the property setting, the picture is deleted from the button and the property setting is again "(none)."

Remarks You can use the Picture property to create a custom tool bar in which each button displays a different picture. You can create the pictures using Microsoft Paintbrush or another application that creates bitmap (.BMP) files. A bitmap file must have a .BMP extension.

Buttons can display either a caption or a picture. If you assign both to a button, the caption is not visible. If the picture is deleted, the caption appears.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

Note When this property is read, the setting is "(none)" or "(bitmap)."

See Also OnPush Property

Pmt Function

Description Returns the payment for an annuity based on periodic, constant payments and a constant interest rate.

Syntax **Pmt**(*rate*, *nper*, *pv*, *fv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **Pmt** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

The arguments *rate* and *nper* must be calculated using payment periods expressed in the same units. For example, if *rate* is calculated using months, *nper* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **FV** Function, **IPmt** Function, **NPer** Function, **PPmt** Function, **PV** Function, **Rate** Function

Example This example returns the payment for a loan over a fixed period. Given are the interest percentage rate per period (`APR / 12`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Const ENDPERIOD = 0, BEGINPERIOD = 1      ' When payments are made.
Const MB_YESNO = 4                        ' Define Yes/No buttons.
Const ID_NO = 7                           ' Define No as a response.
Fmt = "###,###,##0.00"                    ' Define money format.
FVal = 0                                  ' Usually 0 for a loan.
```

```
PVal = InputBox("How much do you want to borrow?")
APR = InputBox("What is the annual percentage rate of your loan?")
If APR > 1 Then APR = APR / 100            ' Ensure proper form.
TotPmts = InputBox("How many monthly payments will you make?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
Payment = Pmt(APR / 12, TotPmts, -PVal, FVal, PayType)
MsgBox "Your payment will be " & Format(Payment, Fmt) & " per month."
```

PopUp Property

Applies To Forms.

Description Determines whether a form opens as a pop-up form.

Setting The PopUp property settings are:

Setting	Description
Yes	The form opens as a pop-up form in Form view. It has thin borders and floats on top of all other Microsoft Access windows.
No	(Default) The form is not a pop-up form.

Remarks You can use the PopUp property to create a custom tool bar or palette similar to the Microsoft Access tool bar and Palette.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

PPmt Function

Description Returns the principal payment for a given period of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **PPmt**(*rate*, *per*, *nper*, *pv*, *fv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **PPmt** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
per	Payment period in the range 1 through *nper*.
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance, is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

The arguments *rate* and *nper* must be calculated using payment periods expressed in the same units. For example, if *rate* is calculated using months, *nper* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **FV** Function, **IPmt** Function, **NPer** Function, **Pmt** Function, **PV** Function, **Rate** Function

Example

This example calculates how much of a payment for a specific period is principal when all the payments are of equal value. Given are the interest percentage rate per period (`APR / 12`), the payment period for which the principal portion is desired (`Period`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Const ENDPERIOD = 0, BEGINPERIOD = 1                ' When payments are made.
Const MB_YESNO = 4                                  ' Define Yes/No buttons.
Const ID_NO = 7                                     ' Define No as a response.
NL = Chr(13) & Chr(10)                              ' Define newline.
TB = Chr(9)                                         ' Define tab.
Fmt = "###,###,##0.00"                              ' Define money format.
FVal = 0                                            ' Usually 0 for a loan.
PVal = InputBox("How much do you want to borrow?")
APR = InputBox("What is the annual percentage rate of your loan?")
If APR > 1 Then APR = APR / 100                     ' Ensure proper form.
TotPmts = InputBox("How many monthly payments do you have to make?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
Payment = Abs(-Pmt(APR / 12, TotPmts, PVal, FVal, PayType))
Msg = "Your monthly payment is " & Format(Payment, Fmt) & "."
Msg = Msg & "Would you like a breakdown of your principal and "
Msg = Msg & "interest per period?"
MakeChart = MsgBox(Msg, MB_YESNO)                   ' See if chart is desired.
If MakeChart <> ID_NO Then
  If TotPmts > 25 Then MsgBox "Only the first 25 periods will be shown."
  Msg = "Month  Payment Principal Interest" & NL
  For Period = 1 To TotPmts
    If Period > 25 Then Exit For                    ' Only show first 25.
    P = PPmt(APR / 12, Period, TotPmts, -PVal, FVal, PayType)
    P = (Int((P + .005) * 100) / 100)               ' Round principal.
    I = Payment - P
    I = (Int((I + .005) * 100) / 100)               ' Round interest.
    Msg = Msg & Period & TB & Format(Payment, Fmt)
    Msg = Msg & TB & Format(P, Fmt) & TB & Format(I, Fmt) & NL
  Next Period
  MsgBox Msg                                        ' Display amortization table.
End If
```

PrimaryKey Property

Applies To Tables.

Description Specifies the names of the field or fields that Microsoft Access uses as the primary key for the table.

Settings In a table's Design view, select the field or fields in the order you want for the primary key and then use the Set Primary Key command from the Edit menu or the Primary Key button on the tool bar. You can also enter one or more field names separated by semicolons.

Remarks Microsoft Access creates an index on the primary key of the table and uses it to find records and to create joins between tables. The primary key index requires an entry in each field of the primary key and allows no duplicates. The order of the fields determines the default sort order for the table.

If there is no primary key when you save a table design, Microsoft Access displays a dialog box asking whether you want a primary key to be created. If you choose Yes, a Counter field is added to the table and set as the primary key. If you choose No, no primary key is created.

A table with no primary key can't be used in a relationship and can be slower to sort and search.

Access Basic Index information is available with the **ListIndexes** method.

See Also Indexed Property, Index1...Index5 Properties

Print Action

Description Prints the active object in the open database. You can print datasheets, reports, and forms.

Action argument	Description
Print Range	The range to print. You can print all of the object (All), the part of the object that is selected (Selection), or a range of pages you specify using the Page From and Page To arguments (Pages). The default is All.
Page From	The first page to print. Printing starts at the top of this page. This argument is required if you select Pages for the Print Range argument.

Action argument	Description
Page To	The last page to print. Printing stops at the bottom of this page. This argument is required if you select Pages for the Print Range argument.
Print Quality	The print quality. You can select High, Medium, Low, or Draft quality. The lower the quality, the faster the object prints. The default is High.
Copies	The number of copies to print. The default is 1.
Collate Copies	Select Yes to collate the printed copies. Select No if you don't need to collate the copies. The object may print faster if this argument is set to No. The default is Yes.

Remarks

This action has the same effect as selecting an object and then choosing the Print command from the File menu.

Tip If you have particular print settings you use frequently, create a macro containing a Print action with these settings in its arguments.

The arguments for this action match the options in the Print dialog box. However, unlike the FindRecord action and Find dialog box, the argument settings aren't shared with the dialog box options.

See Also

OpenForm Action, OpenQuery Action, OpenReport Action, OpenTable Action

Access Basic

Syntax

DoCmd Print [*printrange*] [, *pagefrom*, *pageto*] [, *printquality*] [, *copies*] [, *collatecopies*]

Argument	Description

printrange	One of the following intrinsic constants:
	`A_PRINTALL`
	`A_SELECTION`
	`A_PAGES`
	If you leave this argument blank, the default (`A_PRINTALL`) is assumed.

pagefrom	A numeric expression that is a valid page number in the active form or datasheet. This argument is required if you specify `A_PAGES` for the *printrange* argument.
pageto	A numeric expression that is a valid page number in the active form or datasheet. This argument is required if you specify `A_PAGES` for the *printrange* argument.

Argument	Description

printquality	One of the following intrinsic constants:
	`A_HIGH`
	`A_MEDIUM`
	`A_LOW`
	`A_DRAFT`
	If you leave this argument blank, the default (`A_HIGH`) is assumed.

copies	A numeric expression. If you leave this argument blank, the default (`1`) is assumed.
collatecopies	Use the reserved word **True** (–1) to collate copies and **False** (0) to print without collating. If you leave this argument blank, the default (**True**) is assumed.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example prints two collated copies of the first four pages of the active form or datasheet.

```
DoCmd Print A_PAGES, 1, 4, , 2
```

Print Method

Description Prints text on the **Debug** or **Reports** object using the current color and font.

Syntax *object*.**Print** [{**Spc**(*n*) | **Tab**(*n*)}][*expressionlist*][{; | ,}]

Remarks You can use this method only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

The **Print** method uses these arguments.

Argument	Description
object	**Debug** or **Reports** object on which the text is to be printed.
Spc(*n*)	Name of the Access Basic function optionally used to insert *n* spaces into the printed output. Multiple use is permitted.

Argument	Description
Tab(*n*)	Name of the Access Basic function optionally used to tab to the *n*th column before *expressionlist* is printed. Multiple use is permitted.
expressionlist	Numeric or string expressions to print. If this argument is omitted, **Print** prints a biank line. Multiple expressions can be separated with a space, a semicolon, or a comma. A space has the same effect as a semicolon. You can also use **Spc** and **Tab** to separate each expression.
{; | ,}	Determines the location of the insertion point for the next character displayed: a semicolon means that the insertion point is placed immediately after the last character displayed; a comma means that the insertion point is placed at the start of the next print zone. Print zones begin every 14 columns. The width of each column is an average of the width of all characters in the point size for the chosen font.

The expressions specified by *expressionlist* are printed on the object starting at the position indicated by the CurrentX and CurrentY property settings.

When *expressionlist* is printed, a carriage return is usually appended so that the next **Print** begins printing on the next line. When a carriage return occurs, the CurrentY property setting is increased by the height of *expressionlist* (the same as the value returned by **TextHeight**), and the CurrentX property is set to 0.

When a semicolon follows *expressionlist*, no carriage return is appended, and the next **Print** prints on the same line as the current **Print**. The CurrentX and CurrentY properties are set to the point immediately after the last character printed. If *expressionlist* itself contains carriage returns, each such embedded carriage return sets CurrentX and CurrentY as described above for **Print** without a semicolon.

When a comma follows *expressionlist*, CurrentX and CurrentY are set to the next print zone on the same line.

When *expressionlist* is printed on a **Reports** object, lines that cannot fit in the specified position don't scroll. The text is clipped to fit the object.

When the list is printed on the **Debug** object, up to 255 characters can appear on a single line. If the text is longer, it will scroll to multiple lines.

Because the **Print** method usually prints with proportionally spaced characters, it is important to remember that there is no correlation between the number of characters printed and the number of fixed-width columns those characters occupy. For example, a wide letter (such as W) occupies more than one fixed-width column, whereas a narrow letter (such as I) occupies less. When you use wider-than-average characters, you must ensure that your tabular columns are positioned far enough apart. Alternatively, you can print using a fixed-pitch font (such as Courier) to ensure that each character uses only one column.

See Also CurrentX, CurrentY Properties; **Spc** Function; **Tab** Function

Example This example uses the **Print** method to display text on a report named `Report1`. It uses the **TextWidth** and **TextHeight** methods to center the text vertically and horizontally.

```
Dim Rpt As Report                                   ' Dimension variable.
Set Rpt = Reports!Report1                           ' Assign variable to Report.
Rpt.ScaleMode = 3                                   ' Set scale to pixels.
Rpt.FontName = "Courier"                            ' Use Courier font.
Rpt.FontSize = 24                                   ' Use font size of 24 points.
Message = "DisplayMessage"                          ' Text to be displayed.
HorMes = Rpt.TextWidth(Message)                     ' Horizontal width.
VerMes = Rpt.TextHeight(Message)                    ' Vertical height.
' Calculate position of left and top corner of text to be displayed.
Rpt.CurrentX = Rpt.ScaleWidth / 2 - HorMes / 2
Rpt.CurrentY = Rpt.ScaleHeight / 2 - VerMes / 2
Rpt.Print Message                                   ' Display text.
```

Print # Statement

Description Writes data to a sequential file.

Syntax **Print #** *filenumber*, *expressionlist* [{; | ,}]

Remarks The **Print #** statement uses these arguments.

Argument	Description
filenumber	Number used in an **Open** statement to open a sequential file. It can be any numeric expression that evaluates to the number of an open file. Note that the number sign (#) preceding *filenumber* is not optional.
expressionlist	Numeric and/or string expressions to be written to the file.
{; | ,}	Determines the position of the next character printed. A semicolon means the next character is printed immediately after the last character; a comma means the next character is printed at the start of the next print zone. Print zones begin every 14 columns.

If you omit *expressionlist*, the **Print #** statement prints a blank line in the file.

Because **Print #** writes an image of the data to the file, you must delimit the data so it is printed correctly. If you use commas as delimiters, **Print #** also writes the blanks between print fields to the file.

Also, remember that spacing that works well when data is displayed on a text screen using monospaced characters may not work well when the data is redisplayed in a graphical environment using proportionally spaced characters.

The **Print #** statement usually writes **Variant** data to a file the same way it writes any other Access Basic data type. However, there are some exceptions:

- If the data being written is a **Variant** of **VarType** 0 (Empty), **Print #** writes nothing to the file for that data item.
- If the data being written is a **Variant** of **VarType** 1 (**Null**), **Print #** writes the literal #NULL# to the file.
- If the data being written is a **Variant** of **VarType** 7 (Date), **Print #** writes the date to the file using the Short Date format defined in the WIN.INI file. When either the date or the time component is missing or zero, **Print #** writes only the part provided to the file.

See Also **Open** Statement, **Print** Method, **Spc** Function, **Tab** Function, **Write #** Statement

Example This example uses the **Print #** statement to write data to a test file. Note the use of the comma and semicolon.

```
Open "TESTFILE" For Output As #1                ' Open to write file.
Print #1, "This is a test of the Print # statement."
Print #1,                                       ' Print blank line to file.
Print #1, "Print Zone 1", "Print Zone 2"        ' Print in two print zones.
Print #1, "No space between" ; "!"              ' Print two strings together.
Close #1                                        ' Close file.
```

PrintCount Property

Applies To Reports.

Description Indicates the number of times the OnPrint property setting is evaluated for the current line.

Setting The PrintCount property setting is a **Long** value.

Usage Design view: not available. Other views: read-only.

Remarks Microsoft Access increments the PrintCount property each time the OnPrint property setting is evaluated for the current line. As the next line is printed, Microsoft Access resets PrintCount to 0.

Under some circumstances, the PrintCount property setting might be incremented more than once. If you print a report containing order information, you might keep a running total of the order amounts. The following example shows how you can use the PrintCount property to make sure each order is counted only once.

```
If Reports![My Report].PrintCount = 1 Then
  RunningTotal = RunningTotal + OrderAmount
End If
```

See Also FormatCount Property

PrintSection Property

See MoveLayout, NextRecord, PrintSection Properties

PSet Method

Description Sets a point on a **Reports** object to a specified color.

Syntax *object*.**PSet** [**Step**](*x*, *y*)[, *color*]

Remarks You can use this method only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

The **PSet** method uses these arguments.

Argument	Description
object	**Reports** object on which the point is to be drawn.
Step	Reserved word that specifies that the coordinates are relative to the current graphics position given by the CurrentX and CurrentY property settings of *object*.
x, *y*	**Single** values indicating the horizontal and vertical coordinates of the point to set.
color	RGB color to set the point to. If this argument is omitted, the current foreground color is used. You can use the **RGB** function or the **QBColor** function to specify the color.

The size of the point depends on the DrawWidth property setting. When DrawWidth is 1, **PSet** sets a single pixel to the specified color. When DrawWidth is greater than 1, the point is centered on the specified coordinates.

The way the point is drawn depends on the settings of the DrawMode and DrawStyle properties.

When you apply the **PSet** method, CurrentX and CurrentY are set to the point specified by the arguments.

To clear a single pixel with the **PSet** method, specify the coordinates of the pixel and use white (&HFFFFFF) as the *color* argument.

See Also CurrentX, CurrentY Properties; DrawMode Property; DrawStyle Property; DrawWidth Property; **RGB** Function; **QBColor** Function

Example This example uses the **PSet** method to draw a line through the horizontal axis of a report named `Report1`.

```
Dim Rpt As Report                    ' Dimension variable.
Set Rpt = Reports!Report1            ' Assign variable to Report.
Rpt.ScaleMode = 3                    ' Set scale to pixels.
MidPt = Rpt.ScaleHeight / 2          ' Calculate midpoint.
For I = 1 To Rpt.ScaleWidth          ' Loop to draw line
Rpt.PSet (I, MidPt)                  '  down horizontal axis
Next I                               '  pixel by pixel.
```

Note Using the **Line** method would be a more efficient way to draw this line.

Put Statement

Description Writes from a variable to a disk file.

Syntax **Put** [#] *filenumber*, [*recordnumber*], *variablename*

Remarks The **Put** statement uses these arguments.

Argument	Description
filenumber	Number used in an **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.

Argument	Description
recordnumber	The largest possible *recordnumber* is $2^{31} - 1$, or 2,147,483,647. For files opened in **Random** mode (with the **Random** reserved word), *recordnumber* is the number of the record to be written. For files opened in **Binary** mode (with the **Binary** reserved word), *recordnumber* is the byte position at which writing starts. The first byte in a file is at position 1, the second byte is at position 2, and so on. If you omit *recordnumber*, the next record or byte (the one after the last **Get** or **Put** statement or the one pointed to by the last **Seek** function) is written. And if you omit *recordnumber*, you must still include the delimiting commas. For example: `Put #4,, FileBuffer`
variablename	Name of the variable to write to the file. Any variable can be used except object variables and array variables (variables that describe an entire array). However, you can use a variable that describes a single element of an array.

For files opened in **Random** mode, the following rules apply:

- If the length of the data being written is less than the length specified in the **Len** clause of the **Open** statement, **Put** still writes subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be determined with any certainty, it is generally a good idea to have record length match the length of the data being written.
- If the variable being written is a variable-length string, **Put** writes a 2-byte descriptor containing string length and then the variable. The record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual length of the string.
- If the variable being written is a numeric **Variant** (**VarType** 0–7), **Put** writes 2 bytes identifying the **VarType** of the **Variant** and then the variable. For example, when writing a **Variant** of **VarType** 3, **Put** writes 6 bytes: 2 bytes identifying the **Variant** as **VarType** 3 (**Long**) and 4 bytes containing the **Long** data. The record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual number of bytes required to store the variable.
- If the variable being written is a **String Variant** (**VarType** 8), **Put** writes 2 bytes identifying the **VarType**, 2 bytes indicating the length of the string, and then the string data. The record length specified by the **Len** clause in the **Open** statement must be at least 4 bytes greater than the actual length of the string.

- If the variable being written is any other type of variable (not a variable-length string and not a **Variant**), **Put** writes only the variable data. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the length of the data being written.
- Elements of user-defined types are written using **Put** as if each were written individually except that there is no padding between elements. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the sum of all the bytes required to write the individual elements.

For files opened in **Binary** mode, all of the **Random** rules apply except that:

- In **Binary** mode, the **Len** clause in the **Open** statement has no effect. **Put** writes all variables to disk contiguously, that is, with no padding between records.
- **Put** writes variable-length strings that are not elements of user-defined types without the 2-byte length descriptor. The number of bytes written equals the number of characters in the string. For example, the following statements write 10 bytes to file number 1:

```
VarString$ = String$(10," ")
Put #1,,VarString$
```

See Also **Get** Statement, **Open** Statement

Example This example uses **Put** to write data to a disk file.

```
Dim ClientName As String * 20              ' Declare a fixed-length variable.
Open "TESTFILE" For Random As #1 Len = Len(ClientName$)
ClientName$ = InputBox$("Enter name: ")       ' Get user input.
Do While ClientName$ <> String$(20, " ")
  Put #1, , ClientName$                       ' Write to file.
  ClientName$ = InputBox$("Enter name: ")     ' Get user input.
Loop
Close #1                                      ' Close file.
```

PV Function

Description Returns the present value of an annuity based on periodic, constant payments to be paid in the future and a constant interest rate.

Syntax **PV**(*rate*, *nper*, *pmt*, *fv*, *due*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **PV** function uses the following numeric arguments.

Argument	Description
rate	Interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pmt	Payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance, is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.

The arguments *rate* and *nper* must be calculated using payment periods expressed in the same units. For example, if *rate* is calculated using months, *nper* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

See Also **FV** Function, **IPmt** Function, **NPer** Function, **Pmt** Function, **PPmt** Function, **Rate** Function

Example

This example returns the present value of an $1,000,000 annuity that will provide $50,000 a year for the next 20 years. Provided are the expected annual percentage rate (`APR`), the total number of payments (`20`), the amount of each payment (`YrIncome`), the total future value of the investment (`FVal`), and a number that indicates whether the payment is made at the beginning or end of the payment period (`PayType`). Note that because `YrIncome` represents cash paid out from the annuity each year, it's a negative number.

```
Const ENDPERIOD = 0, BEGINPERIOD = 1        ' When payments are made.
Fmt = "###,##0.00"                          ' Define money format.
APR = .0825                                 ' Annual percentage rate.
TotPmts = 20                                ' Total number of payments.
YrIncome = 50000                            ' Yearly income.
FVal = 1000000                              ' Future value.
PayType = BEGINPERIOD                       ' Payment at beginning of month.
PVal = PV(APR, TotPmts, -YrIncome, FVal, PayType)
MsgBox "The present value is " & Format(PVal, Fmt) & "."
```

QBColor Function

Description

Returns the RGB color code corresponding to a color number.

Syntax

QBColor (*qbcolor*)

Remarks

The *qbcolor* argument is an integer in the range 0 to 15, inclusive, that corresponds to the color values used by other versions of Basic (such as Microsoft QuickBasic™ and Basic Compiler). Starting with the least-significant byte, the returned value specifies the red, green, and blue values used to set the appropriate color in the RGB system used by Access Basic.

The following table shows the possible settings for *qbcolor*.

Setting	Color	Setting	Color
0	Black	8	Gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Yellow	14	Light yellow
7	White	15	Bright white

See Also

RGB Function

Example This example takes a user-specified number and returns the **RGB** value for the equivalent **QBColor** value.

```
Do
  Number = InputBox("Enter a number from 0 through 15.")
Loop Until Number >= 0 And Number <= 15
MsgBox "The RGB number for that color is " & QBColor(Number)
```

Quit Action

Description Exits Microsoft Access. You can select one of several options for saving database objects.

Action argument	Description
Options	Specifies what happens to unsaved objects when you quit Microsoft Access. Select Prompt to display dialog boxes that ask whether to save each object. Select Save All to save all objects without prompting by dialog boxes. Select Exit to quit without saving any objects. The default is Save All.

Remarks Microsoft Access doesn't run any actions that follow the Quit action in a macro.

You can use this action to quit Microsoft Access without prompts from Save dialog boxes by using a custom menu command or a button on a form. For example, you might have a master form that you use to display the objects in your custom workspace. This form could have a Quit button attached to a macro containing the Quit action with the Options argument set to Save All.

This action has the same effect as choosing the Exit command from the File menu. If you have any unsaved objects when you choose this command, the dialog boxes that appear are the same as those displayed when you select Prompt for the Options argument of the Quit action.

See Also StopAllMacros Action, StopMacro Action

Access Basic

Syntax

DoCmd Quit [*options*]

Argument	Description
options	One of the following intrinsic constants: A_PROMPT A_SAVE A_EXIT If you leave this argument blank, the default (A_SAVE) is assumed.

Example

This example displays a dialog box that asks if you want to save any changed objects before you exit Microsoft Access.

```
DoCmd Quit A_PROMPT
```

Randomize Statement

Description Initializes the random-number generator.

Syntax **Randomize** [*number*]

Remarks The argument *number* can be any valid numeric expression. *Number* is used to initialize the random-number generator by giving it a new seed value. If you omit *number,* the value returned by the **Timer** function is used as the new seed value.

If **Randomize** is not used, the **Rnd** function returns the same sequence of random numbers every time the program is run. To have the sequence of random numbers change each time the program is run, place a **Randomize** statement with no argument at the beginning of the program.

See Also **Rnd** Function, **Timer** Function

Example See **Rnd** Function

Rate Function

Description Returns the interest rate per period for an annuity.

Syntax **Rate**(*nper*, *pmt*, *pv*, *fv*, *due*, *guess*)

Remarks An annuity is a series of constant cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **Rate** function uses the following numeric arguments.

Argument	Description
nper	Total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pmt	Payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv	Future value or cash balance you want after you've made the final payment. The future value of a loan, for instance, is $0. As another example, if you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
due	Number indicating when payments are due. Use 0 if payments are due at the end of the payment period, and use 1 if payments are due at the beginning of the period.
guess	Value you guess will be returned by **Rate**. In most cases, *guess* is 0.1 (10 percent).

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Rate is calculated by iteration. Starting with the value of *guess*, **Rate** cycles through the calculation until the result is accurate to within 0.00001 percent. If, after 20 tries, it can't find a result, **Rate** fails. If your guess is 10 percent and **Rate** fails, try a different value for *guess*.

See Also **FV** Function, **IPmt** Function, **NPer** Function, **Pmt** Function, **PPmt** Function, **PV** Function

Example

This example calculates the interest rate of a loan given the total number of payments (`TotPmts`), the amount of the loan payment (`Payment`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`), and an approximation of the expected interest rate (`Guess`).

```
Const ENDPERIOD = 0, BEGINPERIOD = 1      ' When payments are made.
Const MB_YESNO = 4                        ' Define Yes/No buttons.
Const ID_NO = 7                           ' Define No as a response.
Fmt = "##0.00"                            ' Define percentage format.
FVal = 0                                  ' Usually 0 for a loan.
Guess = .1                                ' Guess of 10 percent.
PVal = InputBox("How much did you borrow?")
Payment = InputBox("What's your monthly payment?")
TotPmts = InputBox("How many monthly payments do you have to make?")
PayType = MsgBox("Do you make payments at the end of the month?", MB_YESNO)
If PayType = ID_NO Then PayType = BEGINPERIOD Else PayType = ENDPERIOD
APR = (Rate(TotPmts, -Payment, PVal, FVal, PayType, Guess) * 12) * 100
MsgBox "Your interest rate is " & Format(CInt(APR), Fmt) & " percent."
```

RecordCount Property

Applies To

Recordsets.

Description

Contains the number of records in a recordset.

Setting

The RecordCount property setting is a **Long** value.

Usage

Read-only.

Remarks

As you add and delete table records, the RecordCount value changes to reflect the total number of nondeleted records.

If records are deleted from the underlying tables of a **Dynaset**, the RecordCount value changes to reflect the current number of records and doesn't include the newly deleted records. However, if records are added to the underlying tables, the RecordCount value is not updated to reflect the addition of the new records.

For **Dynasets** and **Snapshots**, the RecordCount value equals the number of records accessed. To determine the total number of records in a **Dynaset** or **Snapshot**, first use the **MoveLast** method and then inspect the RecordCount value.

The total number of records in a **Snapshot** doesn't change even if the number of underlying tables does change.

A recordset with no records has a RecordCount value of 0.

See Also **ListTables** Method

RecordLocks Property

Applies To Forms. Reports.

Description

- Forms—determines how records in the underlying table or query are locked when data in a multiuser database is updated.
- Reports—determines whether records in the underlying table or query are locked while a report is printed or previewed.

Setting The RecordLocks property settings are:

Setting	Description
No Locks	(Default) In forms, a record isn't locked while two or more people are editing it at the same time, but the record locks immediately before the data is saved in the underlying table. In forms, this setting is typically used only to view data or for single-user databases. In reports, no records are locked while the report is previewed or printed.
All Records	All records in the current table or query are locked while the form is open in Form view or while the report is printed or previewed. No one else can edit any of the current records until the form is closed or the report has finished printing. In forms, this setting is typically used for update queries.
Edited Record	(Forms only) A group of records is locked as soon as any user starts editing any field and stays locked until the data is saved to the database. A record can be edited by only one person at a time.

Remarks Use the No Locks setting for forms if only one person uses the database or makes all the changes to data. Use the All Records setting in a form used for an update query to update many records at once. In a report, use the All Records setting when you want to print the data that is current when you start the report. Use the Edited Record setting for forms in a multiuser database when two or more people might edit data at the same time.

Each locked record has a locked symbol in its record selector.

Note To change the default setting for forms, choose the Options command from the View menu. In the Category list, select Multi-User and then set the Default Record Locking option.

Access Basic

The property settings and their values are:

Setting	Value
No Locks	0
All Records	1
Edited Record	2

Design view: read/write. Other views: read-only.

RecordSelectors Property

Applies To

Forms.

Description

Determines whether a form displays record selectors.

Setting

The RecordSelectors property settings are:

Setting	Description
Yes	(Default) Each record has a record selector.
No	No record has a record selector.

Remarks

You can use this property to remove record selectors when you use a form as a dialog box or palette.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also

PopUp Property

RecordSource Property

Applies To Forms. Reports.

Description Specifies the default table, query, or SQL statement on which a form or report is based.

Setting Enter a table name, a query name, or an SQL statement.

Remarks Microsoft Access uses the RecordSource setting to establish which table, query, or set of fields is the basis for a form or report. It then uses the ControlSource setting to bind controls to the fields specified by the RecordSource setting.

You must specify the location within your database of the data that you want to see in a form or report and identify the part of the data to be displayed in each control. Use the RecordSource property to specify where the data is, and use the ControlSource property to specify what appears in each control.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also ControlSource Property; RowSourceType, RowSource Properties

ReDim Statement

Description Used at the procedure level to declare dynamic-array variables and allocate or reallocate storage space.

Syntax **ReDim** [**Preserve**] *variablename*(*subscripts*) [**As** *type*][, *variablename*(*subscripts*) [**As** *type*]] . . .

Remarks The **ReDim** statement uses these arguments.

Argument	Description
Preserve	Preserves the data in an existing array when you change the last dimension.
variablename	Name of a variable.

Argument	Description
subscripts	Dimensions of an array. You can declare multiple dimensions. The syntax of *subscripts* is described below.
As *type*	Reserved word used to declare the data type of a variable. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Variant**, or a user-defined type. Use a separate **As** *type* clause for each variable being defined.

The argument *subscripts* has the following syntax:

[*lower* **To**]*upper*[, [*lower* **To**] *upper*]. . .

The **To** reserved word provides a way to indicate both the lower and upper bounds of an array variable's subscripts. The following statements are equivalent if there is no **Option Base** statement:

```
ReDim A(8,3)
ReDim A(0 To 8, 0 To 3)
ReDim A(8, 0 To 3)
```

Array subscripts can be negative. **To** can be used to specify any range of subscripts between –32,768 and 32,767, inclusive:

```
ReDim A(-4 To 10)
ReDim B(-99 To -5, -3 To 0)
```

You can calculate the amount of memory used by a numeric array by multiplying the number of elements in the array times the number of bytes required by the data type of the array. For example, an **Integer** array, which requires 2 bytes per element, can contain twice as many elements as a **Long** array, which requires 4 bytes per element. An **Integer** array can contain four times as many elements as a **Double** array, which requires 8 bytes per element. **String** arrays are limited to something less than 64K bytes because some small storage overhead is required for each array.

If you use a subscript that is greater than the specified maximum or smaller than the specified minimum, an error occurs and a message is displayed, unless it is trapped in error-handling code. An error also occurs if the size of the array (in terms of bytes of memory used) exceeds the allowable limits described above.

The **ReDim** statement is usually used to size or resize a dynamic array that has already been formally declared using a **Global** or **Dim** statement with empty parentheses (without dimension subscripts.) If you first declare a dynamic array using a **Global** or **Dim** statement and include no dimension subscripts, the maximum number of array dimensions you can later specify with **ReDim** is eight. If your array requires more than eight dimensions, you can use the **ReDim** statement within a procedure to initially declare a local dynamic-array variable. Your array can then have as many as 60 dimensions.

You can use the **ReDim** statement repeatedly to change the number of elements in an array. However, you can't use **ReDim** to change the number of dimensions in an array. For example, if you declare an array consisting of two dimensions (for example, `ReDim A(10,10)`), you can't use **ReDim** later to change it to an array with three dimensions (for example, `ReDim A(12,12,)`). Similarly, you can't declare an array of **Integers** and later use **ReDim** to change the array to another data type.

If you use the **Preserve** reserved word, resize only the last array dimension. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, don't resize the first one. You can change the size of only the last dimension and still preserve the contents of the array. The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
...
ReDim Preserve X(10, 10, 15)
```

The preceding example keeps any data entered in the array `X( )` and adds space for more elements in the third dimension of the array.

Variables are initialized at compile time. Numeric variables are initialized to 0, **Variant** variables to Empty. Variable-length strings are initialized as zero-length strings, and fixed-length strings are filled with ANSI zeros (**Chr**(0)). The fields of user-defined type variables are initialized as if they were separate variables.

See Also **Dim** Statement, **Global** Statement, **Option Base** Statement, **Static** Statement

Example The **ReDim** statement is used to dynamically resize an array while a procedure is running. Note that the array variable is originally defined at the module level using empty parentheses.

```
Dim TestArray() As Integer                ' Declare dynamic array.

Sub ReDim_Demo
  Dim Size As Integer                     ' Declare Integer variable.
  Size = Int(100 * Rnd + 1)               ' Generate random number.
  ReDim TestArray(Size)                   ' Make array have Size elements.
  For I = 1 To Size                       ' Index for number of elements.
    TestArray(I) = Rnd                    ' Put number in each element.
  Next I
  ReDim TestArray(Size * 10)              ' Make array 10 times larger.
  For I = 1 To Size * 10                  ' Index for number of elements.
    TestArray(I) = Rnd                    ' Put number in each element.
  Next I
End Sub
```

Rem Statement

Description Used to include explanatory remarks in a program.

Syntax1 **Rem** *remark*

Syntax 2 ' *remark*

Remarks The argument *remark* is the text of any comments you want to include in your program. Spaces and punctuation are permitted. Comment text appears exactly as entered when the program is listed. You can use comments to document how your code works or to provide any other information with your code.

If you use line numbers or line labels, you can branch from a **GoTo** or **GoSub** statement to a line containing a **Rem** statement. Execution continues with the first executable statement following the **Rem** statement.

As shown in the syntax, you can use a single quotation mark (apostrophe) instead of the **Rem** reserved word. If the **Rem** reserved word follows other statements on a line, it must be separated from the statements by a colon. However, when you use a single quotation mark, the colon is not required after other statements.

Example This example illustrates the various forms of the **Rem** statement syntax.

```
Rem This is the first form of the syntax.
' This is the second form of the syntax.
Msg = "Hello" : Rem Comment after a statement; separated by a colon.
Msg = "Goodbye" ' Second form after a statement; without a colon.
```

Rename Action

Description Renames the selected database object. This object must be selected in the Database window.

Action argument	Description
New Name	A new name for the selected database object. Required argument.

Remarks The new name must follow the standard naming conventions for Microsoft Access objects.

You cannot rename an open object.

You can use the SelectObject action to select the object you want to rename. Set the In Database Window argument of the SelectObject action to Yes.

This action has the same effect as choosing the Rename command from the File menu. The New Name argument takes the place of the Rename dialog box, allowing you to rename an object without having to stop the macro to enter the new name in the dialog box.

This action differs from the CopyObject action, which creates a copy of the object under a new name.

See Also CopyObject Action, SelectObject Action

Access Basic

Syntax

DoCmd Rename *newname*

Argument	Description
newname	A string expression that is the new name for the object selected in the Database window. The name must follow the standard naming conventions for Microsoft Access objects.

Example

This example renames the selected table using the specified name.

```
DoCmd Rename "Old Employees Table"
```

RepaintObject Action

Description Completes any pending screen updates for the specified database object or the active database object if none is specified. Such updates include any pending recalculations for the object's controls.

Action argument	Description
Object Type	The type of object to repaint. Select Table, Query, Form, Report, Macro, or Module from the drop-down list in the Action Arguments section of the Macro window. Leave this argument blank to select the active object.
Object Name	The name of the object to repaint. The drop-down list shows all objects in the database of the type selected with the Object Type argument. If you leave the Object Type argument blank, leave this argument blank also.

Remarks

Microsoft Access sometimes waits to complete pending screen updates until it finishes other tasks. With this action, you can force immediate repainting of the fields in the specified object. You can use this action:

- When you use the SetValue action to change values in a number of fields. Microsoft Access might not show the changes immediately, especially if other fields (such as calculated fields) depend on values in the changed fields.
- When you want to make sure that the form you're viewing displays data in all of its fields. For example, fields containing OLE objects often don't display their data immediately after you open a form.

This action doesn't cause a requery of the database nor does it show new, changed, or deleted records from the object's underlying table or query. Use the Requery action to requery the source of the object or one of its controls. Use the ShowAllRecords action to display the most recent records and remove any applied filters.

Note The RepaintObject action doesn't have quite the same effect as choosing the Refresh command from the Records menu, which shows any changes you or other users have made to the currently displayed records in forms and datasheets.

See Also

Requery Action, SetValue Action, ShowAllRecords Action

Access Basic

Syntax

DoCmd RepaintObject [*objecttype*, *objectname*]

Argument	Description

objecttype	One of the following intrinsic constants:
	`A_TABLE`
	`A_QUERY`
	`A_FORM`
	`A_REPORT`
	`A_MACRO`
	`A_MODULE`

objectname	A string expression that is the valid name of an object of the type selected with the *objecttype* argument.

Remarks

Using the RepaintObject action with no arguments repaints the active window.

Example

This example repaints the form `My Form`.

```
DoCmd RepaintObject A_FORM, "My Form"
```

Report Property

See Form, Report Properties

Reports Object

Description

You use the **Reports** object to refer to a particular report. For example, you can use the **Reports** object to set report properties.

The **Reports** object has no methods and only one property: Count. The following example shows how you can use the Count property to calculate the number of open reports.

```
NumberReports = Reports.Count
```

You can refer to a report in three ways:

- **Reports**![My Report]
- **Reports**("My Report")
- **Reports**(*report number*)

If you use **Reports**(*report number*), notice that the reports are zero-based; that is, the first report is **Reports**(0), the second report is **Reports**(1), and so on.

You can use the **Reports** object in Access Basic and in expressions.

See Also

ActiveReport Property, Count Property, **Debug** Object, **Forms** Object, **Screen** Object

Example

This example uses the **Reports** object to change a report title.

```
Reports![My Report Name]![Report Title].Caption = "New Title Goes Here"
```

The next example displays the name of each report that is currently open.

```
Function List_Reports
  Dim X As Integer
  NumberReports = Reports.Count           ' Count number of reports.
  Debug.Print "There are"; NumberReports; "open reports:"
  For X = 0 To NumberReports - 1
    Debug.Print Reports(X).FormName       ' Print report name.
  Next X
End Function
```

Requery Action

Description Updates the data in a specified control on the active object by requerying the source of the control. If no control is specified, this action requeries the source of the object itself.

Action argument	Description
Control Name	The name of the control you want to update. You should use only the name of the control, not the full syntax (such as **Forms**!*formname*!*controlname*). Leave this argument blank to requery the source of the active object. If the active object is a datasheet or query dynaset, you must leave this argument blank.

Remarks Use this action to ensure that the active object or one of its controls displays the most recent data.

The Requery action does one of the following:

- Reruns the query on which the control or object is based.
- Displays any new, changed, or deleted records from the table on which the control or object is based.

Controls based on a query or table include:

- List boxes and combo boxes.
- Subform controls.
- OLE objects, such as graphs.
- Controls containing aggregate functions, such as **DSum**.

If the specified control is not based on a query or table, this action forces a recalculation of the control.

If you leave the Control Name argument blank, the Requery action has the same effect as pressing SHIFT+F9 when the object has the focus. If a subform control has the focus, this action requeries only the source of the subform (just as pressing SHIFT+F9 does).

Note The Requery action requeries the source of the control or object. In contrast, the RepaintObject action repaints fields in the specified object but doesn't requery the database or display new records. The ShowAllRecords action requeries the active object, but it also removes any applied filters, which the Requery action does not do.

See Also RepaintObject Action, ShowAllRecords Action

Access Basic

Syntax

DoCmd Requery [*controlname*]

Argument	Description
controlname	A string expression that is the name of a control on the active object.

Remarks

Use only the name of the control for the *controlname* argument, not the full syntax.

You can also use a variable dimensioned as a **Control** data type for the *controlname* argument:

```
Dim MyListBox As Control
Set MyListBox = Forms!Form1!Field3
DoCmd Requery MyListBox.ControlName
```

Example

This example uses the Requery action to update the `Employee List` control.

```
DoCmd Requery "Employee List"
```

Reset Statement

Description Closes all disk files.

Syntax **Reset**

Remarks The **Reset** statement closes all open disk files and writes the contents of all file buffers to disk. However, this statement affects only files opened from within Access Basic; it doesn't affect files opened directly by Microsoft Access.

See Also **Close** Statement, **End** Statement

Example This example uses **Reset** to close open files and flush the contents of file buffers out to disk.

```
Open "FILE1" For Random As #1          ' Open one file for Random.
Open "FILE2" For Output As #2          ' Open another file for Output.
...                                    ' Perform some actions on
...                                    ' the open files.
Reset                                  ' Close all files, flush buffers.
```

Restore Action

Description Restores a maximized or minimized window to its previous size.

Remarks This action works on the selected object. If an object has been minimized to an icon, you can first select it using the SelectObject action and then restore it to its previous size using the Restore action.

You can use the MoveSize action to move or size a window that you have restored.

The Restore action has the same effect as clicking the Restore button in the window's upper-right corner or choosing the Restore command from the window's Control menu.

See Also Maximize Action, Minimize Action, MoveSize Action, SelectObject Action

Access Basic

Syntax

DoCmd Restore

Remarks

This action takes no arguments.

Resume Statement

Description Resumes program execution after an error-handling routine is finished.

Syntax **Resume** { [**0**] | **Next** | *line*}

Remarks The various forms of the **Resume** statement redirect program flow as described below.

Statement	Description
Resume [**0**]	Program execution resumes with the statement that caused the error or at the most recently executed call out of the procedure containing the error-handling routine.
Resume Next	Execution resumes with the statement immediately following the one that caused the error or with the statement immediately following the most recently executed call out of the procedure containing the error-handling routine.
Resume *line*	Execution resumes at *line*, which is a line label or line number. The argument *line* must be in the same procedure as the error handler.

Where execution resumes is determined by the location of the error handler in which the error is trapped, not necessarily by the location of the error itself.

The following table summarizes where a program resumes execution when the **Resume** [**0**] statement is used.

Where error occurs	Where program resumes
Same procedure as error handler	Statement that caused the error
Different procedure from error handler	Most recently executed statement in the procedure that contains the error handler

If you use a **Resume** statement anywhere except in an error-handling routine, an error occurs.

When an error-handling routine is active and the end of the procedure (an **End Sub** or **End Function** statement) is encountered before a **Resume** statement is executed, an error occurs because a logical error is presumed to have been made inadvertently. However, if an **Exit Sub** or **Exit Function** statement is encountered while an error handler is active, no error occurs because it is considered a deliberate redirection of program flow.

See Also **On Error** Statement

Example See **On Error** Statement

RGB Function

Description Returns a **Long** value representing an RGB color value.

Syntax **RGB** (*red*, *green*, *blue*)

Remarks The **RGB** function uses these arguments.

Argument	Description
red	Integer in the range 0 to 255, inclusive, that represents the red component of the color
green	Integer in the range 0 to 255, inclusive, that represents the green component of the color
blue	Integer in the range 0 to 255, inclusive, that represents the blue component of the color

Access Basic methods and properties that accept a color specification expect that specification to be a **Long** value that represents an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue, which taken together cause a specific color to be displayed.

If the value of any argument to **RGB** exceeds 255, it is assumed to be 255.

The following table shows the RGB values of some standard colors and the red, green, and blue components of each.

Color	RGB value	Red value	Green value	Blue value
Black	&H0	0	0	0
Blue	&HFF0000	0	0	255
Green	&HFF00	0	255	0
Cyan	&HFFFF00	0	255	255
Red	&HFF	255	0	0
Magenta	&HFF00FF	255	0	255
Yellow	&HFFFF	255	255	0
White	&HFFFFFF	255	255	255

See Also **QBColor** Function

Example See **Line** Method

Right, Right$ Functions

Description Return the rightmost *n* characters of a string argument.

Syntax **Right**[**$**](*stringexpr*, *n*)

Remarks **Right** returns a **Variant**; **Right$** returns a **String**.

The argument *stringexpr* can be any string expression. However, only **Right** can accept a **Variant** of **VarType** 1 (**Null**) as *stringexpr*, in which case, a **Null** is returned.

The argument *n* is a **Long** expression indicating how many characters to return. It must be between 0 and approximately 65,535, inclusive. If *n* is 0, the return value is a zero-length string. If *n* is greater than or equal to the number of characters in *stringexpr*, the entire string is returned.

To find the number of characters in *stringexpr*, use **Len**(*stringexpr*).

See Also **Left**, **Left$** Functions; **Len** Function; **Mid**, **Mid$** Functions

Example This example returns the two rightmost characters from a variable called `MyText`.

```
= Right(MyText, 2)
```

RmDir Statement

Description Removes an existing directory.

Syntax **RmDir** *path*

Remarks The argument *path* is a string expression that specifies the name of the directory to be removed. This argument must have fewer than 128 characters and has the following syntax:

[*drive*:] [\ ]*directory*[*directory*] . . .

The argument *drive* is an optional drive specification; the argument *directory* is a directory name.

The **RmDir** statement works like the operating system command **rmdir**. If the directory to be removed contains anything except the working directory ('.') and the parent directory ('..'), an error occurs.

See Also **ChDir** Statement; **CurDir**, **CurDir$** Functions; **MkDir** Statement

Example See **MkDir** Statement

Rnd Function

Description Returns a random number.

Syntax **Rnd**[(*number*)]

Remarks The argument *number* can be any valid numeric expression.

Rnd returns a **Single** value less than 1 but greater than or equal to 0.

The value of *number* determines how **Rnd** generates a random number.

Value of *number*	Number returned
< 0	The same number every time, as determined by *number*
> 0	The next random number in the sequence
= 0	The number most recently generated
number omitted	The next random number in the sequence

The same random-number sequence is generated every time the program is run because each successive call to the **Rnd** function uses the previous random number as a seed for the next number in the random-number sequence.

To have the program generate a different random-number sequence each time it is run, use the **Randomize** statement without an argument to initialize the random-number generator before **Rnd** is called.

To produce random integers in a given range, use this formula:

Int((*upperbound* – *lowerbound* + 1) * **Rnd** + *lowerbound*)

Here, *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

See Also **Abs** Function, **Atn** Function, **Cos** Function, Derived Math Functions, **Exp** Function, **Fix** Function, **Int** Function, **Log** Function, **Randomize** Statement, **Sgn** Function, **Sin** Function, **Sqr** Function, **Tan** Function

Example This example simulates rolling a pair of dice by generating random values from 1 to 6. Each time this program is run, **Randomize** uses the **Timer** function to generate a new random-number sequence.

```
Randomize                               ' Seed random-number generator.
Dice1 = Int(6 * Rnd + 1)                ' Generate first die value.
Dice2 = Int(6 * Rnd + 1)                ' Generate second die value.
```

Rollback Statement

Description Reverses changes made during the current transaction and ends the transaction.

Syntax **Rollback**

Remarks A transaction is a series of changes you make to a recordset. You mark the beginning of a transaction with the **BeginTrans** statement. You use the **Rollback** or **CommitTrans** statement to end a transaction.

You can use **Rollback** to undo and **CommitTrans** to save changes made during the current transaction. After you make changes to your database, you can decide whether to undo the changes or commit (save) them. If you nest a transaction, you must roll back or save the current transaction before you can roll back or save one at an earlier level.

You can nest up to five levels of transactions by using multiple **BeginTrans** statements. Once you start a transaction, use additional **BeginTrans** statements to isolate smaller groups of changes. You can then roll back or save changes to these smaller groups instead of to an entire large transaction.

Once you save a transaction, you can't use **Rollback** to undo the changes made since the most recent **BeginTrans** statement (unless the changes are part of a larger transaction that is rolled back).

After you use **Rollback**, the record that was current before you used the corresponding **BeginTrans** statement becomes the current record again.

If you use **Rollback** without a matching **BeginTrans** statement, an error occurs.

Caution If a recordset is based entirely on Microsoft Access tables, its Transactions property is **True** (–1) and you can use transactions. Tables created by other database products, however, may not support transactions. For example, you can't use transactions in a recordset based on a Paradox table. In this case, the Transactions property is **False** (0). You should inspect the Transactions property before using the **Rollback** statement to make sure that the recordset supports transactions and rolling back. When used with a nonsupported recordset, **Rollback** is ignored; no error occurs.

Note Transactions in Access Basic are independent of the transactions performed in Microsoft Access forms and reports.

See Also **BeginTrans** Statement, **CommitTrans** Statement, Transactions Property

Example This example changes the job title of all sales representatives in the Employees table of the NWIND.MDB database. It uses **BeginTrans** to start a transaction that isolates all of the changes made to the Employees table. **Rollback** can be used to undo changes saved with the **Update** method.

```
Const MB_ICONQUESTION = 32
Const YES = 6
Const YES_NO = 4
Dim CRLF As String, EmployeeName As String
Dim Message As String, Prompt As String
Dim MyDB As Database, MyTable As Table
CRLF = Chr$(13) + Chr$(10)
Prompt = "Change title to Account Executive?"
Set MyDB = CurrentDB()                          ' Get current database.
Set MyTable = MyDB.OpenTable("Employees") ' Open Table.
```

```
BeginTrans                                              ' Begin transaction.
Do Until MyTable.EOF
  If MyTable!Title = "Sales Representative" Then
    EmployeeName = MyTable("Last Name") + ", " + MyTable("First Name")
    Message = "Employee: " + EmployeeName + CRLF + CRLF
    If MsgBox(Message + Prompt, MB_ICONQUESTION + YES_NO, "Change Job
    ➥ Title") = YES Then
      MyTable.Edit                                      ' Enable editing.
      MyTable!Title = "Account Executive" ' Change title.
      MyTable.Update                                    ' Save changes.
    End If
  End If
  MyTable.MoveNext                                      ' Move to next record.
Loop
If MsgBox("Save all changes?", MB_ICONQUESTION + YES_NO, "Save Changes")
➥ = YES Then
  CommitTrans                                           ' Commit changes.
Else
  Rollback                                              ' Undo changes.
End If
MyTable.Close                                           ' Close Table.
```

Tips

- Although you can use a loop to inspect each record in a **Table** or **Dynaset**, using an update query might be more efficient. The following example shows how you can change the sales representatives' job titles without using a **Do...Loop** statement.

```
Dim MyDB As Database, MyQuery As QueryDef
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQuery("Change Job Titles")       ' Create query.
' Set SQL property.
MyQuery.SQL = "UPDATE DISTINCTROW Employees SET Employees!Title =
➥ WHERE Employees!Title = 'Sales Representative';"
MyQuery.Execute                                           ' Invoke query.
MyQuery.Close                                             ' Close query.
MyDB.DeleteQuery ("Change Job Titles")                    ' Delete query.
```

- You can also choose the Replace command from the Edit menu. Using the Replace command is probably faster than running an update query.

RowSource Property

See RowSourceType, RowSource Properties

RowSourceType, RowSource Properties

Apply To Controls (combo box, graph, list box, unbound object frame [embedded]).

Description

- RowSourceType—specifies the type of source of the object's data.
- RowSource—specifies the source of the object's data.

Setting The RowSourceType property settings are:

Setting	Description
Table/Query	(Default) The data is from a table or is the result of a query or an SQL statement.
Value List	The data is a list of items specified by the RowSource setting.
Field List	The data is a list of field names from the table or query named by the RowSource setting.

You can also enter the name of an Access Basic function that defines the list. Enter only the function's name; do not include an equal sign (=), parentheses, or arguments.

The RowSource property settings depend on the source type specified by RowSourceType.

Type	Setting
Table/Query	Enter the name of a table, a query, or an SQL statement.
Value List	Enter a list of items separated by semicolons.
Field List	Enter the name of a table or query.

Remarks These properties work together to tell Microsoft Access how to provide data to a list box, a combo box, or an OLE object.

If RowSourceType is set to Value List, the following are valid entries for the RowSource property for a list box or combo box:

Mon;Tue;Wed
One-column list with three rows (ColumnCount = 1).

Yes;No
One-column list with two rows (ColumnCount = 1).

1;Monday;2;Tuesday;3;Wednesday;4;Thursday
Two-column list with four rows (ColumnCount = 2).

Country;Capital;China;Beijing;Brazil;Brasilia
Two-column list with two rows of data (ColumnCount = 2, ColumnHeads = Yes).

If RowSourceType is Table/Query, the following is a valid SQL statement for a two-column list drawn from the Categories table in the Northwind Traders sample database:

```
SELECT [Category ID], [Category Name] FROM [Categories]
➥ ORDER BY [Category Name];
```

Tip If you can't select the first row of a list box in Form view, the ColumnHeads property is probably set to Yes.

Access Basic

To set the RowSourceType property, use a string expression with one of these values: `"Table/Query"`, `"Value List"`, or `"Field List"`. Use a string expression to set the value of the RowSource property.

Design view for both properties: read/write. Other views for both properties: read-only.

See Also

BoundColumn Property, ColumnWidths Property, ControlSource Property, RecordSource Property

RSet Statement

Description

Right aligns a string within the space of a string variable.

Syntax

RSet *stringvar* = *stringexpr*

Remarks

The **RSet** statement uses these arguments.

Argument	Description
stringvar	Name of a **String** variable
stringexpr	String expression to be right aligned within *stringvar*

If *stringexpr* is shorter than *stringvar*, **RSet** right aligns *stringexpr* within *stringvar*. **RSet** replaces any leftover characters in *stringvar* with spaces, back to its beginning.

If *stringexpr* is longer than *stringvar*, **RSet** places only the leftmost characters, up to the length of *stringvar*, in *stringvar*. Characters beyond the length of *stringvar* are truncated from the right.

You can't use **RSet** to assign variables of one user-defined type to those of another.

See Also

LSet Statement

Example This example uses **RSet** to right align some text in a **String** variable named `Text2` within `Field1` (another **String** variable).

```
Dim Field1 As String * 25                ' Create a fixed-length variable.
Text2 = "17-character text"              ' Create a 17-character text.
RSet Field1 = Text2                      ' Right align the 17-character text
                                         '  within the 25-character space.
```

RTrim, RTrim$ Functions

See **LTrim**, **LTrim$**, **RTrim**, **RTrim$**, **Trim**, **Trim$** Functions

RunApp Action

Description Runs a Microsoft Windows-based or MS-DOS application, such as Microsoft Excel, Microsoft Word for Windows, or Microsoft PowerPoint® from within Microsoft Access.

Action argument	Description
Command Line	Command line used to start the application (including the path name and any other necessary parameters, such as switches that run the application in a particular mode). Required argument.

Remarks The application selected with this action loads and runs in the foreground. The macro containing this action continues to run.

You can use this action to run a program that you need to use with Microsoft Access. For example, you may want to use Microsoft Excel to have spreadsheet data pasted into or linked to your Microsoft Access database.

You can transfer data between the other application and Microsoft Access by using the Windows dynamic data exchange (DDE) facility or the Clipboard. You can use the SendKeys action to send keystrokes to the other application.

MS-DOS applications run in an MS-DOS window within the Windows environment.

This action is similar to using the Access Basic **Shell** function.

This action has the same effect as double-clicking a program name in a Windows program group or running a program from the File Run dialog box in the Windows Program Manager.

Access Basic This action isn't available from Access Basic. Use the **Shell** function to run another application.

RunCode Action

Description Executes an Access Basic **Function** procedure.

Action argument	Description
Function Name	The name of the Access Basic **Function** procedure to execute. Put any function arguments in parentheses. Required argument.

Remarks The user-defined **Function** procedures are stored in Microsoft Access modules.

Microsoft Access ignores the return value of the function.

Tip To execute a **Sub** procedure written in Access Basic, create a **Function** procedure that calls the **Sub** procedure. Then use the RunCode action to execute the **Function** procedure.

See Also **Eval** Function

Access Basic This action isn't available from Access Basic. Run the desired function directly in Access Basic.

RunMacro Action

Description Runs a macro. The macro can be in a macro group.

Action argument	Description
Macro Name	The name of the macro to run. The drop-down list in the Action Arguments section of the Macro window shows all macros (and macro groups) in the current database. Enter *macrogroupname.macroname* to run a particular macro in a macro group. Required argument.
Repeat Count	The maximum number of times the macro will run. If you leave this argument blank (and Repeat Expression is also blank), the macro runs once.
Repeat Expression	An expression that evaluates to true or false. The macro stops running if the expression evaluates to false. The expression is evaluated each time the macro runs.

Remarks

If you enter a macro group name for the Macro Name argument, Microsoft Access runs the first macro in the macro group.

You can use this action to:

- Call a macro from within another macro.
- Run a macro based on a certain condition.
- Attach a macro to a custom menu command.

This action is similar to choosing the Run Macro command from the File menu, which runs a specified macro. However, the command runs the macro only once, whereas the RunMacro action runs a macro as many times as you want (up to 65,535 times).

Tip You can use the Repeat Count and Repeat Expression arguments to determine how many times the macro will run:

- If you leave both arguments blank, the macro runs once.
- If you enter a number for Repeat Count but leave Repeat Expression blank, the macro runs the specified number of times.
- If you leave Repeat Count blank but enter an expression for Repeat Expression, the macro runs until the expression evaluates to false.
- If you enter values for both arguments, the macro runs the number of times specified in Repeat Count or until Repeat Expression evaluates to false, whichever occurs first.

Access Basic

Syntax

DoCmd RunMacro *macroname* [, *repeatcount*] [, *repeatexpression*]

Argument	Description
macroname	A string expression that is the valid name of a macro in the current database.
repeatcount	A numeric expression that evaluates to an integer, which is the number of times the macro will run.
repeatexpression	A numeric expression that is evaluated each time the macro runs. When it evaluates to **False** (0), the macro stops running.

Remarks

You can use *"macrogroupname.macroname"* for the *macroname* argument to run a particular macro in a macro group.

If you specify the *repeatexpression* argument and leave the *repeatcount* argument blank, you must include the *repeatcount* argument's comma. If you leave a trailing argument blank, don't use a comma following the last argument you specify.

Example

This example runs the macro `My Macro`.

```
DoCmd RunMacro "My Macro"
```

The following example triggers the OnPush event of `Button1` on `Form1`.

```
Dim B As Control
Set B = Forms!Form1!Button1
If (Left(B.OnPush, 1) = "=") Then
  Temp = Eval(Mid(B.OnPush,2))
Else
  DoCmd RunMacro B.OnPush
End If
```

RunningSum Property

Applies To Control (text box) on a report.

Description Determines whether a text box displays a running total and specifies the range over which the values are accumulated.

Setting The RunningSum property settings are:

Setting	Description
No	(Default) The text box displays the current value.
Over Group	The text box displays a running sum of values in the same group level. The value accumulates until the next higher group level section is encountered.
Over All	The text box displays a running sum of values in the same group level. The value accumulates until the end of the report.

Remarks Use the RunningSum property to have Microsoft Access automatically calculate any of three different kinds of totals for each group level in a report.

You can have up to ten nested group levels in a report. See the example, which shows a sales report with two group levels.

See Also OnFormat Property

Example This example shows a sales report with two group levels: Month and Type of Food. Each text box in the group footer section (January Chocolates, January Meat, and so on) contains the expression "=Sum ([Sales])" with the RunningSum property set to one of the three different values.

Group level	No.	Over group	Over all
Month: January			
Type of Food: Chocolates			
02-Jan	25		
22-Jan	75		
January Chocolates	100	Jan Food 100	YTD Sales 100
Type of Food: Meat			
03-Jan	90		
20-Jan	60		
January Meat	150	Jan Food 250	YTD Sales 250
Month: February			
Type of Food: Chocolates			
05-Feb	70		
19-Feb	80		
February Chocolates	150	Feb Food 150	YTD Sales 400
Type of Food: Meat			
12-Feb	150		
22-Feb	100		
February Meat	250	Feb Food 400	YTD Sales 650

Access Basic The property settings and their values are:

Setting	Value
No	0
Over Group	1
Over All	2

Design view: read/write. Other views: read-only.

RunSQL Action

Description Runs a Microsoft Access action query using the corresponding SQL statement.

Action argument	Description
SQL Statement	The SQL statement for the action query you want to run. The maximum length of this statement is 256 characters. Required argument.

Remarks You use action queries to append, delete, and update records and to save a query's dynaset as a new table. With the RunSQL action, you can perform these operations directly from a macro without having to use stored queries.

Microsoft Access queries are actually SQL queries that you design and run using the Query window. The following table shows the Microsoft Access action queries and their corresponding SQL statements.

Action query	SQL statement
Append query	INSERT INTO
Delete query	DELETE
Make-table query	SELECT...INTO
Update query	UPDATE

You can also use the IN clause with these statements to modify data in another database.

Note To run a select query from a macro, use the View argument in the OpenQuery action to open an existing select query in Datasheet view. You can also run existing action queries in the same way.

Tip To see the SQL equivalent of a Microsoft Access query, choose the SQL command from the View menu in the Query window. Use the listed SQL statements as models to create SQL queries to run with the RunSQL action. Duplicating an SQL query in the SQL Statement argument for the RunSQL action has the same effect as running this Microsoft Access query from the Query window.

See Also OpenQuery Action

Access Basic

Syntax

DoCmd RunSQL *sqlstatement*

Argument	Description
sqlstatement	A string expression that is a valid SQL statement for an action query. It uses an INSERT INTO, DELETE, SELECT...INTO, or UPDATE statement. Include an IN clause if you want to access another database.

Example

This example updates an Employees table, changing each sales manager's title to `Regional Sales Manager`.

```
DoCmd RunSQL "UPDATE Employees SET Employees.Title = ""Regional Sales
➥ Manager"" WHERE Employees.Title = ""Sales Manager"";"
```

Scale Method

Description Defines the coordinate system for a **Reports** object.

Syntax *object*.**Scale** [(*x1*, *y1*) – (*x2*, *y2*)]

Remarks You can use this method only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

The **Scale** method uses these arguments.

Argument	Description
object	**Reports** object to which the coordinate system is applied.
x1, y1	Single-precision values for the horizontal and vertical coordinates that define the position of the upper-left corner of the object.
x2, y2	Single-precision values for the horizontal and vertical coordinates that define the position of the lower-right corner of the object.

You can use the **Scale** method to reset the coordinate system to any scale you choose. Using **Scale** with no arguments resets the coordinate system to twips. **Scale** affects the coordinate system for the print and graphics methods.

See Also ScaleHeight, ScaleWidth Properties; ScaleLeft, ScaleTop Properties; ScaleMode Property

Example See **Circle** Method

ScaleHeight, ScaleWidth Properties

Apply To Reports.

Description Specify the number of units for the horizontal (ScaleWidth) or vertical (ScaleHeight) measurement of the current section when the **Circle**, **Line**, **PSet**, and **Print** methods are used while a report is printed or previewed.

These properties can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

Setting Use a numeric expression to set the value of each property. The default settings are the internal height and width of a section in twips.

Usage Design view: not available. Other views: read/write.

The value of this property is a **Single** data type.

Remarks You can use these properties to create a custom coordinate scale for drawing or printing. For example, the statement `ScaleHeight = 100` defines the internal height of the section as 100 units, or one vertical unit as 1/100 of the height.

Use the ScaleMode property to define a scale based on a standard unit of measurement, such as twips, points, pixels, characters, inches, millimeters, or centimeters.

Setting these properties to positive values makes coordinates increase in value from top to bottom and left to right. Setting them to negative values makes coordinates increase in value from bottom to top and right to left.

Using these properties and the related ScaleLeft and ScaleTop properties, you can set up a full coordinate system with both positive and negative coordinates. All four of these scale properties interact with the ScaleMode property in the following ways:

- Setting any other scale property to any value automatically sets ScaleMode to 0.
- Setting ScaleMode to a number greater than 0 changes ScaleHeight and ScaleWidth to the new unit of measurement and sets ScaleLeft and ScaleTop to 0. Also, the CurrentX and CurrentY property settings change to reflect the new coordinates of the current point.

You can also use the **Scale** method to set the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties in one statement.

Note The ScaleHeight and ScaleWidth properties aren't the same as the Height and Width properties.

See Also DrawMode Property, DrawStyle Property, FillColor Property, ForeColor Property

ScaleLeft, ScaleTop Properties

Apply To Reports.

Description Set the horizontal (ScaleLeft) or vertical (ScaleTop) coordinates that describe the location of the left and top edges of a page when the **Circle**, **Line**, **PSet**, and **Print** methods are used while a report is previewed or printed or its output is saved to a file.

These properties can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

Setting Use a numeric expression to set the value of these properties. The default for each property is 0.

Usage Design view: not available. Other views: read/write.

The value of each property is a **Single** data type.

Remarks Using these properties and the related ScaleHeight and ScaleWidth properties, you can set up a full coordinate system with both positive and negative coordinates. These four scale properties interact with the ScaleMode property in the following ways:

- Setting any other scale property to any value automatically sets ScaleMode to 0.
- Setting ScaleMode to a number greater than 0 changes ScaleHeight and ScaleWidth to the new unit of measurement and sets ScaleLeft and ScaleTop to 0. Also, the CurrentX and CurrentY property settings change to reflect the new coordinates of the current point.

You can also use the **Scale** method to set the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties in one statement.

Note The ScaleLeft and ScaleTop properties aren't the same as the Left and Top properties.

See Also DrawMode Property, DrawStyle Property, FillColor Property, ForeColor Property,

ScaleMode Property

Applies To Reports.

Description Sets the unit of measurement for coordinates on a page when the **Circle**, **Line**, **PSet**, and **Print** methods are used while a report is previewed or printed.

This property can be used only in an Access Basic function or a macro attached to one of the event properties, such as OnFormat.

Setting

Use a numeric expression to set the value of this property. The ScaleMode property settings are:

Setting	Description
0	Indicates that one or more of the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties are set to custom values
1	(Default) Twip (1440 twips per inch; 567 twips per centimeter)
2	Point (72 points per inch)
3	Pixel (smallest unit of monitor resolution)
4	Character (horizontal = 120 twips per unit; vertical = 240 twips per unit)
5	Inch
6	Millimeter
7	Centimeter

Usage

Design view: not available. Other views: read/write.

The value of this property is an **Integer** data type.

Remarks

Using the related ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties, you can create a custom coordinate system with both positive and negative coordinates. These four scale properties interact with the ScaleMode property in the following ways:

- Setting the value of any other scale property to any value automatically sets ScaleMode to 0.
- Setting ScaleMode to a number greater than 0 changes ScaleHeight and ScaleWidth to the new unit of measurement and sets ScaleLeft and ScaleTop to 0. The CurrentX and CurrentY property settings change to reflect the new coordinates of the current point.

See Also

DrawMode Property, DrawStyle Property, FillColor Property, ForeColor Property, **Scale** Method

ScaleTop Property

See ScaleLeft, ScaleTop Properties

ScaleWidth Property

See ScaleHeight, ScaleWidth Properties

Scaling Property

Applies To Controls (bound object frame, graph, unbound object frame).

Description Determines how Microsoft Access displays the contents of an object frame or a graph.

Setting The Scaling property settings are:

Setting	Description
Clip	(Default) Displays as much of an object's contents as possible with no size adjustments
Scale	Displays the entire contents after adjusting it to fill the height and width of the graph or object frame with possible distortion of the proportions
Zoom	Displays the entire contents after adjusting it to fill either the height or width of the graph or object frame without distortion of the proportions

Remarks Use the Clip setting for the fastest display. You can use the Scale setting for bar graphs and line graphs without concern for size adjustments. The Scale setting can cause distortion in circles and photos.

Access Basic The property settings and their values are:

Setting	Value
Clip	0
Scale	1
Zoom	2

Design view: read/write. Other views: read-only.

Screen Object

Description You use the **Screen** object to refer to a form, control, or report. For example, you can use the **Screen** object to refer to the active form without knowing which particular form it is.

Referring to the **Screen** object doesn't make a form, control, or report active. To make a form, control, or report active, use the SelectObject action.

The **Screen** object has no methods and three properties: ActiveControl, ActiveForm, and ActiveReport.

You can use the **Screen** object in Access Basic and in expressions.

See Also ActiveControl Property, ActiveForm Property, ActiveReport Property, **Debug** Object, **Forms** Object, **Reports** Object, SelectObject Action

Example This example uses the **Screen** object to refer to the active form.

```
MsgBox "The " & Screen.ActiveForm.Caption & " form is active."
```

ScrollBars Property

Applies To Forms. Tool and control (text box) on a form.

Description Determines whether scroll bars appear on a form or text box and whether navigation buttons appear on a form.

Setting To see the property sheet, choose the Properties button on the tool bar or the Properties command from the View menu. Select the form, tool, or control.

The ScrollBars property settings are:

Setting	Description
Neither	(Default for text box.) No scroll bars or navigation buttons on form.
Horizontal Only	Horizontal scroll bar and navigation buttons on form. Not applicable to text boxes.
Vertical Only	Vertical scroll bar but no navigation buttons on form.
Both	(Default for forms) Vertical and horizontal scroll bars and navigation buttons on form. Not applicable to text boxes.

Remarks Changing the default property settings affects only the current form. To change default values for all new forms, you must create a new template.

Access Basic

The property settings and their values are:

Setting	Value
Neither	0
Horizontal Only	1
Vertical Only	2
Both	3

Design view: read/write. Other views: read-only.

See Also Modal Property, PopUp Property, RecordSelectors Property

Second Function

Description Returns an integer between 0 and 59, inclusive, that represents the second of the minute for a time argument.

Syntax **Second**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Hour** Function, **Minute** Function, **Month** Function, **Now** Function, **Weekday** Function, **Year** Function

Example See **Minute** function

Section Property

Applies To Forms. Reports. Controls (all).

Description

- Forms and reports—identifies a section of a form or report and provides access to other properties of that section.
- Controls—identifies the section of a form or report on which the control appears.

Setting The Section property setting is a number that identifies a particular section.

Setting	Description
0	Detail section
1	Form or report header section
2	Form or report footer section
3	Form or report page header section
4	Form or report page footer section
5	Group-level 1 header section (reports only)
6	Group-level 1 footer section (reports only)
7	Group-level 2 header section (reports only)
8	Group-level 2 footer section (reports only)

If a report has additional group-level sections, the header/footer pairs are numbered consecutively beginning with 9.

Usage Design view: read-only. Other views: read-only.

Remarks

Forms and reports

You cannot use the Section property by itself; you must combine it with the properties that apply to form or report sections. The following example shows how you can refer to the Visible property of the page header section of `My Form`.

```
Forms![My Form].Section(3).Visible
```

If you refer to a section that doesn't exist, an error occurs. For example, you might create a form that doesn't have a form header section.

Controls

You can use the Section property to determine in which section a control appears on a form or report. The following example uses the Section property to determine which section contains `My Control`.

```
X = Forms![My Form]![My Control].Section
```

See Also CanGrow, CanShrink Properties; FillColor Property; ForceNewPage Property; Height, Width Properties; KeepTogether Property; NewRowOrCol Property; OnFormat Property; OnPrint Property; SpecialEffect Property; Visible Property

Seek Function

Description Returns the current file position.

Syntax **Seek**(*filenumber*)

Remarks The argument *filenumber* is the number used in the **Open** statement to open the file. You can use any numeric expression that evaluates to the number of an open file.

Seek returns a value between 1 and 2,147,483,647 (equivalent to $2^{31}-1$), inclusive. For files open in **Random** mode, **Seek** returns the number of the next record read or written. For files opened in **Binary**, **Output**, **Append**, or **Input** mode, **Seek** returns the byte position at which the next operation is to take place. The first byte in a file is at position 1, the second byte is at position 2, and so on.

See Also **Get** Statement, **Open** Statement, **Put** Statement, **Seek** Statement

Example In this example, the **Seek** statement sets the file position, and the **Seek** function returns the current file position.

```
Dim NameField As String * 20              ' Declare record variable.
' Create new data file and fill with three records.
Open "TESTFILE" For Random As #1 Len = Len(NameField)
' Get user input to fill records.
For I = 1 To 3
  NameField = InputBox$("Enter student name: ")
  Put #1, I, NameField                    ' Put record on disk.
Next I
Close #1                                  ' Close data file.

' Open test data file.
Open "TESTFILE" For Random As #1 Len = Len(NameField)
Max = LOF(1) \ Len(NameField)             ' Calculate # of records.
For I = Max To 1 Step -1                  ' Read file backwards.
  Seek #1, I                              ' Set file position.
  Get #1, , NameField                     ' Get record.
  Msg = "Record #" & (Seek(1) - 1) & " contains: "
  Msg = Msg & NameField
  MsgBox Msg                              ' Display student name.
Next I
Close #1                                  ' Close file.
```

Seek Method

Description Locates the record in an indexed table that satisfies the specified criteria for the current index and makes that record the current record.

Syntax *table*.**Seek** *comparison, key1, key 2. . .*

Remarks The **Seek** method uses the following arguments.

Argument	Description
table	Name of an existing **Table** object
comparison	One of the following string expressions: "<", "<=", "=", ">=", or ">"
key1, key2. . .	One or more of the **Table's** PrimaryKey or Index1...Index5 property values

Use the **Seek** method to locate a record in a **Table** that satisfies a specific condition. To locate records in a **Dynaset** or **Snapshot** that satisfy a specific condition, use the Find methods. To include all records, not just those that satisfy a specific condition, use the Move methods to move from record to record.

Warning If you edit the current record, be sure you save the changes with the **Update** method before you move to another record. If you move without updating, your changes will be lost, and no error will occur.

Seek searches through the specified key fields and locates the first record that satisfies the criteria specified by *comparison* and *key1*. It makes that record current.

You must set the current index with the Index property before you use **Seek**. If the index identifies a nonunique key field, **Seek** locates the first record that satisfies the criteria.

If *comparison* is "=", ">=", or ">", **Seek** starts at the beginning of the index. If *comparison* is "<=" or "<", **Seek** starts at the end of the index and searches backwards.

The *key1* argument must be of the same field data type as the current index. For example, if the current index refers to a numeric key field (such as Employee ID), *key1* must be numeric. Similarly, if the current index refers to a Text field (such as Last Name), *key1* must be a string.

There doesn't have to be a current record when you use **Seek**.

You can use the **ListIndexes** method to obtain a list of existing indexes.

If *table* doesn't refer to an open table, or if there is no current index, an error occurs.

Note Always inspect the value of NoMatch to determine whether the **Seek** method has succeeded. If it fails, NoMatch will be **True** (–1) and the current record will be undefined.

See Also BOF Property; EOF Property; **FindFirst**, **FindLast**, **FindNext**, **FindPrevious** Methods; Index Property; **ListIndexes** Method; **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious** Methods; NoMatch Property

Example This example opens the Products table in the NWIND.MDB database and uses **Seek** to locate the first record containing a value of 1 in the Supplier ID field (which is a non-unique key field). It changes 1 to 2 and saves the change with **Update**. Subsequent passes through the loop locate the next record that satisfies the condition.

```
Dim MyDB As Database, MyTable As Table
Set MyDB = CurrentDB()
Set MyTable = MyDB.OpenTable("Products")  ' Open Table.
MyTable.Index = "Supplier ID"             ' Define current index.
Do Until MyTable.NoMatch
  MyTable.Seek "=", 1                     ' Seek record.
  If Not MyTable.NoMatch Then             ' If record was found.
    MyTable.Edit                          ' Enable editing.
    MyTable("Supplier ID") = 2            ' Change Supplier ID.
    MyTable.Update                        ' Save changes.
  End If
Loop                                      ' End of loop.
MyTable.Close                             ' Close Table.
```

Using an update query to change Supplier ID values might be more efficient. The next example changes the same Supplier ID values without using a **Do...Loop**.

```
Dim MyDB As Database, MyQuery As QueryDef
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("Change Supplier ID")      ' Create query.
' Set SQL property.
MyQuery.SQL = "UPDATE DISTINCTROW Products SET Products![Supplier ID] = 2
➥ WHERE Products![Supplier ID] = 1;"
MyQuery.Execute                           ' Invoke query.
MyQuery.Close                             ' Close query.
MyDB.DeleteQueryDef ("Change Supplier ID")                   ' Delete query.
```

Tip You can also choose the Replace command from the Edit menu. Using the Replace command is probably quicker than running an update query.

Seek Statement

Description Sets the position in a file for the next read or write.

Syntax **Seek** [#]*filenumber*, *position*

Remarks The **Seek** statement uses these arguments.

Argument	Description
filenumber	Number used in the **Open** statement to open the file. It can be any numeric expression that evaluates to the number of an open file.
position	Numeric expression that indicates where the next read or write should occur. The value of *position* must be between 1 and 2,147,483,647 (equivalent to $2^{31}-1$), inclusive.

For files opened in **Random** mode, *position* is the number of a record in the file.

For files opened in **Binary**, **Input**, **Output**, or **Append** mode, *position* is the byte position relative to the beginning of the file. The first byte in a file is at position 1, the second byte is at position 2, and so on. After a **Seek** operation, the next file input/output operation starts at the specified byte.

Note Record numbers in **Get** and **Put** statements override file positioning done by **Seek**.

Performing a file write after doing a **Seek** operation beyond the end of a file extends the file. If you attempt a **Seek** operation to a negative or zero position, an error occurs.

See Also **Get** Statement, **Open** Statement, **Put** Statement, **Seek** Function

Example See **Seek** Function

Select Case Statement

Description Executes one of several statement blocks depending on the value of an expression.

Syntax

Select Case *testexpression*
  [**Case** *expressionlist1*
    [*statementblock-1*]]
  [**Case** *expressionlist2*
    [*statementblock-2*]]
  [**Case Else**
    [*statementblock-n*]]
End Select

Remarks The **Select Case** statement uses these arguments.

Argument	Description
testexpression	Any numeric or string expression.
statementblock	The elements *statementblock-1* to *statementblock-n* consist of any number of statements on one or more lines.
expressionlist	These elements can take any of the following three forms: ■ *expression* [, *expression*] ■ *expression* **To** *expression* ■ **Is** *comparison-operator expression*

The argument *expressionlist* has these parts.

Part	Description
expression	Any numeric or string expression. The type of the expression must be compatible with the type of *testexpression*. (The type of the expression will be coerced to the same type as *testexpression*. For example, if *testexpression* is an integer, *expressionlist* can contain a double-precision data type.)
comparison-operator	Any valid comparison operator, except **Like**.

If *testexpression* matches the *expressionlist* associated with a **Case** clause, the statement block following that **Case** clause is executed up to the next **Case** clause, or for the final one, up to the **End Select**. Control then passes to the statement following **End Select**.

If you use the **To** reserved word to indicate a range of values, the smaller value must precede **To**.

You can use a comparison operator only in conjunction with the **Is** reserved word.

If **Case Else** is used, its associated statements are executed only if *testexpression* doesn't match any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen *testexpression* values. When there is no **Case Else** statement and no expression listed in the **Case** clauses matches *testexpression*, program execution continues at the statement following **End Select**.

You can use multiple expressions or ranges in each **Case** clause. For example, the following line is valid:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

You also can specify ranges and multiple expressions for character strings. In the following example, **Case** matches strings that are exactly equal to `everything`, strings that fall between `nuts` and `soup` in alphabetical order, and the current value of `TestItem$`:

```
Case "everything", "nuts" To "soup", TestItem$
```

If *testexpression* matches more than one **Case** clause, only the statements following the first match are executed.

Select Case statements can be nested. Each **Select Case** statement must have a matching **End Select** statement.

Note The **Option Compare** statement affects string comparison.

See Also **If...Then...Else** Statement, **On...GoSub** Statement, **On...GoTo** Statement, **Option Compare** Statement

Example This example uses **Select Case** to select between alternative user input.

```
X = InputBox("Please enter a number from 1 to 5.")
Select Case Val(X)                          ' Test input.
  Case 1 To 3                               ' See if less than 4.
    Msg = "You entered a number less than 4."
  Case 4                                    ' Equal to 4.
    Msg = "You entered 4."
  Case Else                                 ' Must be something else.
    Msg = "You entered 5 or another number."
End Select
MsgBox Msg                                  ' Display message.
```

SelectObject Action

Description Selects the specified database object.

Action argument	Description
Object Type	The type of database object to select. Select Table, Query, Form, Report, Macro, or Module from the drop-down list in the Action Arguments section of the Macro window. Required argument.
Object Name	The name of the object to select. The drop-down list shows all objects in the database of the type selected with the Object Type argument. Required argument.
In Database Window	Determines whether Microsoft Access selects the object in the Database window. Select Yes or No from the drop-down list. The default is No.

Remarks The SelectObject action works with any Microsoft Access object that can receive the focus. This action gives the specified object the focus and shows the object if it is hidden. If the object is a form, the SelectObject action sets the form's Visible property to Yes and returns the form to the mode set by its form properties (for example, a modal or pop-up form).

If the object isn't open in one of the other Microsoft Access windows, you can select it in the Database window by setting the In Database Window argument to Yes. If you set In Database Window to No, an error message will appear when you try to select an object that isn't open.

Often, you might use this action to select an object on which you want to perform additional actions. For example, you may want to restore an object that has been minimized (using the Restore action) or maximize a window that contains an object you want to work with (using the Maximize action).

If you select a form, you can use the GoToControl, GoToRecord, and GoToPage actions to move to specific areas on the form. The GoToRecord action also works for datasheets.

Tips

- If you leave the Object Name argument blank and select Yes for the In Database Window argument, Microsoft Access moves the focus to the Database window and chooses the button for the type of object specified by the Object Type argument to display the list of all the objects of this type in the database.
- If you select Yes for the In Database Window argument, you can use the CopyObject or Rename action to copy or rename any of the objects in the database.

See Also CopyObject Action, Rename Action, Restore Action, SetValue Action

Access Basic

Syntax

DoCmd SelectObject *objecttype*, *objectname* [, *indatabasewindow*]

Argument	Description

objecttype	One of the following intrinsic constants:
	`A_TABLE`
	`A_QUERY`
	`A_FORM`
	`A_REPORT`
	`A_MACRO`
	`A_MODULE`

objectname	A string expression that is the valid name of an object of the type selected with the *objecttype* argument.
indatabasewindow	Use the reserved word **True** (–1) to select the object in the Database window. Use the reserved word **False** (0) to select an object that is already open. If you leave this argument blank, the default (**False**) is assumed.

Remarks

If you leave the *indatabasewindow* argument blank, don't use a comma following the *objectname* argument.

Example

This example selects the form `My Form` in the Database window.

```
DoCmd SelectObject A_FORM, "My Form", True
```

SendKeys Action

Description Sends keystrokes directly to Microsoft Access or to an active Microsoft Windows-based application.

Action argument	Description
Keystrokes	The keystrokes you want Microsoft Access or the application to process. You can type up to 255 characters. Required argument.
Wait	Specifies whether the macro should pause until the keystrokes have been processed. Select Yes or No. The default is No.

Remarks Microsoft Access processes the keystrokes it receives through the SendKeys action exactly as if you had typed them directly in a Microsoft Access window.

To specify the keystrokes, use the same syntax as the **SendKeys** statement in Access Basic.

Often, you'll use this action to enter information in a dialog box, particularly if you don't want to interrupt the macro to respond to the dialog box. Some Microsoft Access actions, such as Print and FindRecord, automatically select the options in certain frequently used dialog boxes. You can use the SendKeys action to select the options in less commonly used dialog boxes. For example, you could enter an object's new name (and press the ENTER key) in the Rename dialog box that appears when you use the Rename action.

Note Because the dialog box suspends the macro, you must put the SendKeys action before the action that causes the dialog box to open and set the Wait argument to No.

If you want to send more than 255 characters to Microsoft Access or another Windows-based application, you can use several SendKeys actions in succession in a macro.

See Also GoToControl Action, SelectObject Action, SendKeys Statement, SetValue Action

Access Basic This action isn't available from Access Basic. Use the **SendKeys** statement.

SendKeys Statement

Description Sends one or more keystrokes to the active window as if they had been entered at the keyboard.

Syntax **SendKeys** *keytext* [,*wait*]

Remarks The **SendKeys** statement uses these arguments.

Argument	Description
keytext	String expression sent to the active window.
wait	Numeric expression. If *wait* is true (nonzero), keystrokes must be processed before control is returned to the procedure. If *wait* is false (0), control is returned to the procedure immediately after the keys are sent. If *wait* is omitted, false (0) is assumed as the default. You can use the keywords **True** (–1) and **False** (0) to supply a value for *wait*.

Each key is represented by one or more characters. To specify a single keyboard character, use the character itself. For example, to represent the letter A, use "A" for *keytext*. If you want to represent more than one character, append each additional character to the one preceding it. To represent the letters A, B, and C, use "ABC" for *keytext*.

The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses ( ) have special meanings to **SendKeys**. To specify one of these characters, enclose it inside braces. For example, to specify the plus sign, use {+}. Brackets ([ ]) have no special meaning to **SendKeys**, but you must enclose them in braces as well, because in other applications for Microsoft Windows brackets do have special meaning that may be significant when dynamic data exchange (DDE) occurs. To send brace characters, use {{} and {}}.

To specify characters that aren't displayed when you press a key (such as ENTER or TAB) and keys that represent actions rather than characters, use the codes shown below.

Key	Code	Key	Code
BACKSPACE	{BACKSPACE} or {BS} or {BKSP}	BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}	CLEAR	{CLEAR}
DEL	{DELETE} or {DEL}	DOWN ARROW	{DOWN}
END	{END}	ENTER	{ENTER} or ~
ESC	{ESCAPE} or {ESC}	HELP	{HELP}
HOME	{HOME}	INS	{INSERT}
LEFT ARROW	{LEFT}	NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}	PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}	RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}	TAB	{TAB}
UP ARROW	{UP}	F1	{F1}
F2	{F2}	F3	{F3}
F4	{F4}	F5	{F5}
F6	{F6}	F7	{F7}
F8	{F8}	F9	{F9}
F10	{F10}	F11	{F11}
F12	{F12}	F13	{F13}
F14	{F14}	F15	{F15}
F16	{F16}		

To specify keys combined with any combination of SHIFT, CTRL, and ALT keys, precede the regular key code with one or more of the following codes.

Key	Code
SHIFT	+
CTRL	^
ALT	%

To specify that SHIFT, CTRL, and/or ALT should be held down while several other keys are pressed, enclose the keys' code in parentheses. For example, to have the SHIFT key held down while E and C are pressed, use "+(EC)". To have SHIFT held down while E is pressed, followed by C being pressed without SHIFT, use "+EC".

To specify repeating keys, use the form `{key number}`; you must put a space between `key` and `number`. For example, `{LEFT 42}` means press the LEFT ARROW key 42 times; `{h 10}` means press h 10 times.

SendKeys can't send keystrokes to an application that is not designed to run in the Microsoft Windows environment.

See Also **DoEvents** Function, **DoEvents** Statement

Example This example uses **Shell** to leave the current application and run the Calculator application included with Microsoft Windows; it then uses the **SendKeys** statement to send keystrokes to either quit the Calculator or add some numbers, depending on the user's choice.

```
X = Shell("Calc.exe", 1)              ' Shell Calculator.
For I = 1 To 100                      ' Set up counting loop.
  SendKeys I & "{+}", True            ' Send keystrokes to Calculator
Next I                                '  to add each value of I.
SendKeys "=", True                    ' Get grand total.
AppActivate "Microsoft Access - Jon King" ' Return focus to Microsoft Access.
Msg = "Choose OK to close the Calculator." ' Stop to see results.
MsgBox Msg
AppActivate "Calculator"              ' Return focus to Calculator.
SendKeys "%{F4}", True                ' Send Alt+F4 to close Calculator.
```

Set Statement

Description Assigns an object reference to a variable.

Syntax **Set** *variablename* = *objectexpr*

Remarks The **Set** statement uses the following arguments.

Argument	Description
variablename	Name of the variable to which the object reference is being assigned
objectexpr	Expression consisting of the name of an object, another declared variable of the same object data type, or a function or method that returns an object

To be valid, *variablename* must have been formally declared (using a **Dim**, **Global**, or **Static** statement) to be of an object data type consistent with the object being assigned to it. Valid object data types are **Control**, **Database**, **Dynaset**, **Form**, **QueryDef**, **Report**, **Table**, and **Snapshot**.

The following example illustrates how **Dim** is used to formally declare the variable `View1` as a **Dynaset**. **Set** then assigns a reference to the **Dynaset** (created using the **CreateDynaset** method on the Customer database) to the `View1` variable.

```
Dim View1 As Dynaset
Set View1 = Customer.CreateDynaset("ActiveClients")
```

When you use **Set** to assign an object reference to a variable, no copy of the object is created for that variable. Instead, a reference to the object is created. More than one object variable can refer to the same object. Because these variables are references to rather than copies of the object, any change in the object is also a change for all variables that refer to it.

Note Use the **Let** statement to assign the value displayed by a control to a variable.

See Also **Dim** Statement, **Global** Statement, **Static** Statement

Example This example uses **Set** to assign references to two Microsoft Access objects to variables that have been formally declared using **Dim**. A reference to the current database is assigned to `DB`, and a reference to the Employees table in the current database is assigned to `MyTable`.

```
Dim DB As Database                          ' Dimension database.
Dim MyTable As Table                        ' Dimension table.
Set DB = CurrentDB()                        ' Create database reference.
Set MyTable = DB.OpenTable("Employees")     ' Create table reference.
```

SetValue Action

Description Sets the value of a Microsoft Access field, control, or property on a form, a form datasheet, or a report.

Action argument	Description
Item	The name of the field, control, or property whose value you want to set. Use the full syntax to refer to this item, such as *fieldname* (for a field on the form or report from which the macro was called) or **Forms**!*formname*!*fieldname*. Required argument.
Expression	The expression Microsoft Access uses to set the value for this item. Always use the full syntax to refer to any objects in the expression. For example, to increase the value in a Salary field on an Employees form by 10 percent, use **Forms**!Employees!Salary * 1.1. Required argument. **Note** You don't have to use the equal sign before the expression in this argument. If you do, Microsoft Access evaluates the expression, and then uses this value as the expression in this argument. This can produce unexpected results if the expression is a string. For example, if you put ="String1" in this argument, Microsoft Access first evaluates this expression as String1. Then it uses String1 as the expression in this argument, expecting to find a control or property named String1 on the form or report that called the macro.

Remarks You can use this action to set a value for a field on a form, a form datasheet, or a report. You can also set the values for the Visible, Enabled, and Locked properties of fields on a form in Form view (the view in which these three properties are usually set), and for most properties of controls on a form or report in Design view. See the Access Basic information in each property's topic to determine whether you can use the SetValue action to set the property in Design view.

You can also set the value for a field in a form's underlying table even if the field isn't on the form. Use the syntax **Forms**!*formname*!*fieldname* for the Item argument to set the value for such a field.

If the field whose value you are setting is on a form, the SetValue action doesn't trigger the field's validation rule but does trigger recalculation. The recalculation may not happen immediately. To trigger immediate repainting and force the recalculation to completion, use the RepaintObject action.

You can use the SetValue action in a macro attached to the AfterUpdate property but not in a macro attached to the BeforeUpdate property.

Note You can't use the SetValue action to set the value of the following controls:

- Bound controls and calculated controls on reports
- Calculated controls on forms

Tip You can use the SetValue action to hide or show a form in Form view. Enter **Forms!***formname*.Visible for the Item argument and No or Yes for the Expression argument. Setting a modal form's Visible property to No hides the form and makes it modeless. Setting the property to Yes displays the form and makes it modal again.

See Also RepaintObject Action, SendKeys Action

Access Basic This action isn't available from Access Basic. Set the value directly in Access Basic.

SetWarnings Action

Description Turns system messages on or off.

Action argument	Description
Warnings On	Specifies whether system messages are displayed. Select Yes or No. The default is No.

Remarks You can use this action to prevent modal warnings and message boxes from stopping the macro. However, error messages always appear.

This action has the same effect as pressing ENTER whenever one of the warnings or message boxes appears. Typically, an OK or Yes is chosen in response to the warning or message.

When the macro finishes, Microsoft Access automatically turns the display of the system messages back on.

Often, you'll use this action with the Echo action, which hides the results of a macro until it is finished. You can use the SetWarnings action to hide the warnings and message boxes as well.

Caution Although the SetWarnings action can simplify interactions with macros, you must be careful about turning system messages off. In some situations, you won't want to continue a macro if a certain warning message appears. Unless you're confident of the outcome of all macro actions, you should avoid using this action.

See Also Echo Action

Access Basic

Syntax

DoCmd SetWarnings *warningson*

Argument	Description
warningson	Use the reserved word **True** (–1) to turn on the display of system messages and **False** (0) to turn it off.

Example

This example turns the display of system messages off.

```
DoCmd SetWarnings False
```

Sgn Function

Description Returns a value indicating the sign of a number.

Syntax **Sgn**(*number*)

Remarks The argument *number* can be any valid numeric expression. Its sign determines the value returned by the **Sgn** function:

- If *number* > 0, then **Sgn**(*number*) returns 1.
- If *number* = 0, then **Sgn**(*number*) returns 0.
- If *number* < 0, then **Sgn**(*number*) returns –1.

See Also **Abs** Function, **Atn** Function, **Cos** Function, Derived Math Functions, **Exp** Function, **Fix** Function, **Int** Function, **Log** Function, **Rnd** Function, **Sin** Function, **Sqr** Function, **Tan** Function

Example This example uses **Sgn** to determine the sign of a number.

```
Number = InputBox("Enter a number.")      ' Get user input.
Select Case Sgn(Number)                   ' Evaluate user input.
  Case 0                                  ' Zero.
    Msg = "You entered zero."
  Case 1                                  ' Positive.
    Msg = "You entered a positive number."
  Case -1                                 ' Negative.
    Msg = "You entered a negative number."
End Select
MsgBox Msg                                ' Display results.
```

Shell Function

Description Runs an executable program.

Syntax **Shell**(*commandstring* [, *windowstyle*])

Remarks The **Shell** function uses these arguments.

Argument	Description
commandstring	Includes the name of the program to execute and any required arguments or command line switches. If the program name in *commandstring* doesn't include a .COM, .EXE, .BAT, or .PIF file extension, an error will occur.
windowstyle	Determines the style of the window in which the program is to be executed. If *windowstyle* is omitted, the program is opened minimized with focus, the same as *windowstyle* = 2.

The following table identifies the values for *windowstyle* and the resulting style of window.

Value	Window style
1, 5, 9	Normal with focus
2	Minimized with focus
3	Maximized with focus
4, 8	Normal without focus
6, 7	Minimized without focus

If the **Shell** function successfully executes the named program, it returns the task identification (ID) of the started program. The task ID is a unique number that identifies the running program. If the **Shell** function can't start the named program, an error occurs and a message is displayed, unless it is trapped by error-handling code.

Example This example uses **Shell** to leave the current application and run the Calculator application included with Microsoft Windows; it then uses the **SendKeys** statement to send keystrokes to either quit the Calculator or add some numbers, depending on the user's choice. **AppActivate** is used to move the focus between Microsoft Access and the Calculator.

```
AppActivate "Microsoft Access - Jon King" ' Change focus to Microsoft Access.
SendKeys "%{ }{Down 3}{Enter}", True      ' Minimize Microsoft Access.
X = Shell("Calc.exe", 1)                  ' Shell Calculator.
For I = 1 To 100                          ' Set up counting loop.
  SendKeys I & "{+}", True                ' Send keystrokes to Calculator
Next I                                    '  to add each value of I.
```

```
SendKeys "=", True ' Get grand total.
AppActivate "Microsoft Access - Jon King" ' Return focus to Microsoft Access.
Msg = "Choose OK to close the Calculator."
MsgBox Msg
AppActivate "Calculator"                  ' Return focus to Calculator.
SendKeys "%{F4}", True                    ' Send Alt+F4 to close Calculator.
AppActivate "Microsoft Access - Jon King" ' Return focus to Microsoft Access.
SendKeys "%{ }{Enter}", True              ' Restore Microsoft Access to size.
```

ShowAllRecords Action

Description

Removes any applied filter from the active form and displays all records in the form's underlying table or query.

Remarks

Use this action to ensure that all records (including any changed or new records) are displayed for a form. This action causes a requery of a form's or subform's records.

You can also use this action to remove a filter that was applied with the ApplyFilter action, the Apply Filter/Sort command on the Records menu, or the Filter Name or Where Condition argument of the OpenForm action.

This action has the same effect as choosing the Show All Records command from the Records menu in Form view. This command is also available in a table's or query's Datasheet view. The ShowAllRecords action, however, can be used only in Form view.

See Also

ApplyFilter Action, OpenForm Action, RepaintObject Action, Requery Action

Access Basic

Syntax

DoCmd ShowAllRecords

Remarks

This action takes no arguments.

ShowGrid Property

Applies To Forms in Datasheet view.

Description Specifies whether columns and rows appear with gridlines when the form is displayed in Datasheet view.

Setting To set the ShowGrid property in Datasheet view, choose the Gridlines command from the Layout menu.

From Access Basic, the ShowGrid property values are:

Setting	Description
True (nonzero)	Gridlines appear in the datasheet.
False (0)	No gridlines are visible in the datasheet.

Usage Design view: read/write. Other views: read-only.

Sin Function

Description Returns the sine of an angle.

Syntax **Sin**(*angle*)

Remarks The argument *angle* can be any valid numeric expression measured in radians.

Sin takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

The result lies in the range –1 to 1.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi. Pi approximately equals 3.141593.

See Also **Atn** Function, **Cos** Function, **Tan** Function

Example This example uses **Sin** to calculate the sine of an angle with a user-specified number of degrees.

```
Pi = 4 * Atn(1)                                          ' Calculate Pi.
Degrees = InputBox("Enter an angle in degrees.")         ' Specify angle.
Radians = Degrees * (Pi / 180)                           ' Convert to radians.
Sine = Sin(Radians)                                      ' Calculate sine.
MsgBox "The sine of this angle is " & Sine & "."
```

Single Data Type

Description **Single** (single-precision floating-point) variables are stored as 32-bit numbers (4 bytes) ranging in value from –3.402823E38 to –1.401298E–45 for negative values, from 1.401298E–45 to 3.402823E38 for positive values, and 0. The type-declaration character for a **Single** is **!** (ANSI character 33).

Each floating-point value consists of three parts: the sign, the exponent, and the mantissa. In a single-precision number, the sign takes 1 bit, the exponent takes 8 bits, and the mantissa uses the remaining 23 bits and an additional implied bit.

See Also Access Basic Data Types, **Double** Data Type

SLN Function

Description Returns the straight-line depreciation of an asset for a single period.

Syntax **SLN**(*cost*, *salvage*, *life*)

Remarks The **SLN** function uses the following numeric arguments.

Argument	Description
cost	Initial cost of the asset
salvage	Value of the asset at the end of its useful life
life	Length of the useful life of the asset

The period for which the depreciation is returned is expressed in the same unit of time as the useful life of the asset. All arguments must be positive numbers.

See Also **DDB** Function, **SYD** Function

Example This example returns the straight-line depreciation of an asset for a single period given the asset's initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), and the total life of the asset in years (`LifeTime`).

```
Const YEARMONTHS = 12                          ' Number of months in a year.
Fmt = "###,##0.00"                             ' Define money format.
InitCost = InputBox("What's the initial cost of the asset?")
SalvageVal = InputBox("What's the asset's value at the end of
➥ its useful life?")
MonthLife = InputBox("What's the asset's useful life in months?")
```

```
Do While MonthLife < YEARMONTHS          ' Ensure period is >= 1 year
  MsgBox "Asset life must be a year or more."
  MonthLife = InputBox("What's the asset's useful life in months?")
Loop
LifeTime = MonthLife / YEARMONTHS        ' Convert months to years.
If LifeTime <> Int(MonthLife / YEARMONTHS) Then
  LifeTime = Int(LifeTime + 1)           ' Round up to nearest year.
End If
PDepr = SLN(InitCost, SalvageVal, LifeTime)
MsgBox "The depreciation is " & Format(PDepr, Fmt) & " per year."
```

Sort Property

Applies To **Dynasets**, **Snapshots**.

Description Sets the order for records in a **Dynaset** or **Snapshot**.

Setting The Sort property setting is the ORDER BY clause in an SQL string (without the words ORDER BY).

Usage Read/write.

Remarks When you use this property, sorting occurs when the **Dynaset** or **Snapshot** is created and overrides any sorting specified in a **QueryDef**.

The default sort order is ascending. The following examples show how you can set the sort order to ascending and descending.

```
MySet.Sort = "[Ship Country] Asc"
MySet.Sort = "[Ship Country] Desc"
```

The Sort property doesn't apply to **Tables**. To sort on a table index, use the Index property.

You can't use the Sort property with **Snapshots** created with the **ListFields**, **ListIndexes**, **ListParameters**, or **ListTables** method.

See Also Filter Property, Index Property

Example This example uses the Sort property to set the sort order of a **Dynaset** based on the Orders table of the NWIND.MDB database. The records in `SortedSet` will be in ascending Ship Country order.

```
Dim MyDB As Database, MySet As Dynaset, SortedSet As Dynaset
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("Orders")  ' Create Dynaset.
MySet.Sort = "[Ship Country]"             ' Set sort order.
Set SortedSet = MySet.CreateDynaset()     ' Create second Dynaset.
```

Tip It might be more efficient to create the second **Dynaset** with the desired conditions in one step. As a general rule, when you know which data you want to select, it is usually better to create the **Dynaset** with an SQL string. The next example shows how you can create just one **Dynaset** and obtain the same results as in the preceding example.

```
Dim MyDB As Database, MySet As Dynaset
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset(SELECT * FROM Orders ORDER BY [Ship Country];)
```

SourceObject Property

Applies To Controls (subform/subreport, unbound object frame [linked]).

Description

- Subform/subreport—identifies the subform or subreport on a form or report.
- Linked object frame—contains the fully qualified file name of the file that contains the data linked to the frame.

Setting Enter the name of the subform or subreport, or enter the name of the file you want to link to the frame.

Remarks You can insert a subform or subreport by entering its name in the SourceObject property setting on the property sheet of the main form or report. In Form view and Print Preview, Microsoft Access displays or prints the subform or subreport as if it were part of the main form or report.

To unlink a subform or subreport, erase the SourceObject property setting. This causes the subform or subreport to become blank but does not remove the subform or subreport control. To remove the control in Design view, select the control and then press the DEL key.

See Also LinkChildFields, LinkMasterFields Properties; Parent Property

Access Basic

The property setting is a **String** value.

The following example displays the name of a subform.

```
Debug.Print Forms![My Form]![My Subform].SourceObject
```

Design view: read-only. Other views: read-only.

Space, Space$ Functions

Description Return a string consisting of a specified number of spaces.

Syntax **Space**[$](*number*)

Remarks **Space** returns a **Variant**; **Space$** returns a **String**.

The argument *number* specifies the number of spaces you want in the string. It can be of any numeric data type but is rounded to a **Long** and must be between 0 and approximately 65,535, inclusive.

See Also **Spc** Function; **String**, **String$** Functions

Example This example uses **Space** to create a **Variant** called `Pad` containing 10 spaces.

```
Pad = Space(10)
```

Spc Function

Description Skips a specified number of spaces in a **Print #** statement or **Print** method.

Syntax **Spc**(*number*)

Remarks This function can be used only in Access Basic code and only with the **Print #** statement or the **Print** method.

The argument *number* is an integer between 0 and 32,767, inclusive, that determines the number of blank characters to print.

Note that the **Spc** function does more than move to a new print position. The *number* argument specifies how many blank or space characters are printed starting at the current print position. When you use **Spc** on a line with other arguments to either the **Print #**

statement or the **Print** method and you place no separator (a semicolon or comma) after it, a semicolon (;) is assumed except when **Spc** appears at the end of a line. For example, `Print #1, Spc(10) FixLen1$` and `Print#1, Spc(10); FixLen1$` are equivalent.

When used with files, **Spc** behaves in the following way:

- If *number* is greater than the output-line width, **Spc** calculates *printposition* = *number* **Mod** *width* and generates the number of blanks indicated by that calculation, starting at the current print position.
- If the difference between the current print position and the output-line width is less than *number* (or *number* **Mod** *width*), the **Spc** function skips to the beginning of the next line and generates a number of blanks equal to:

  number – (*width* – *currentprintposition*)

When you use the **Print** method with a proportionally spaced font, the width of space characters printed using the **Spc** function is always an average of the width of all characters in the point size for the chosen font. However, there is no correlation between the number of characters printed and the number of fixed-width columns those characters occupy. For example, the uppercase letter W occupies more than one fixed-width column and the lowercase letter I occupies less. Make sure your tabular columns are wide enough to accommodate wider letters.

See Also **Print** Method; **Print #** Statement; **Space, Space$** Functions; **Tab** Function; **Width #** Statement

Example This example uses the **Spc** function to embed 10 spaces in an output file.

```
Open "TESTFILE" For Output As #1          ' Create test file.
Print #1, "This is a test of the "; Spc(10); "Spc function."
Close #1                                  ' Close file.
```

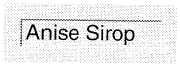
SpecialEffect Property

Applies To Form sections. Report sections. Tools and controls (all except command buttons, lines, page breaks, and subforms/subreports).

Description Determines whether a section or control appears flat, raised, or sunken.

Setting Set this property using the property sheet or the Palette.

The SpecialEffect property settings are:

Setting	Example	Description
Normal		(Default) The object appears flat and has the default colors or custom colors that were set in Design view.
Raised		The object has a highlight on the top and left and a shadow on the bottom and right.
Sunken		The object has a shadow on the top and left and a highlight on the bottom and right.

Note On some displays, such as VGA monochrome and EGA, the Raised and Sunken settings may not affect the appearance of controls or sections unless you have changed the ButtonText color specified in the Microsoft Windows version 3.1 Control Panel from white to a color other than gray.

Access Basic

The property settings and their values are:

Setting	Value
Normal	0
Raised	1
Sunken	2

Design view: read/write. Other views: read-only.

See Also BorderColor Property, BorderStyle Property, BorderWidth Property, BackStyle Property

SQL Property

Applies To **QueryDefs**.

Description Sets the selection criteria for a query. This property can be used only in Access Basic code.

Setting The SQL property setting is an SQL string.

Usage Read/write.

Remarks The SQL property determines how records are selected when you use the query. You can use the query to select records for a **Dynaset** or **Snapshot** or to update a recordset.

For a query that has parameters, the SQL string contains variables that you set prior to each execution of the query. If the query has no parameters, the same conditions are applied each time you use the query.

See Also **CreateDynaset** Method, **CreateSnapshot** Method, Filter Property, **OpenQueryDef** Method, Sort Property

Example This example creates a parameter query based on the Orders table in the NWIND.MDB database. `FindAmount` is a Currency parameter set to `1000`, but a different `FindAmount` value can be specified at run time, either by the user or as the result of a calculation elsewhere in the program.

```
Dim MyDB As Database, MyQuery As QueryDef, MySnapshot As Snapshot
Set MyDB = CurrentDB()
Set MyQuery = MyDB.CreateQueryDef("Large Orders")          ' Create query.
MyQuery.SQL = "Parameters FindAmount Currency; SELECT * FROM Orders
➥  WHERE [Order Amount] > FindAmount;"
MyQuery!FindAmount = 1000                                  ' Set parameter.
Set MySnapshot = MyDB.CreateSnapshot("Large Orders")       ' Create Snapshot.
```

Sqr Function

Description Returns the square root of a number.

Syntax **Sqr**(*number*)

Remarks The argument *number* can be any valid numeric expression that results in a value greater than or equal to 0.

Sqr returns a **Double**.

See Also **Abs** Function, **Atn** Function, **Cos** Function, Derived Math Functions, **Exp** Function, **Fix** Function, **Int** Function, **Log** Function, **Rnd** Function, **Sgn** Function, **Sin** Function, **Tan** Function

Example This example uses **Sqr** to calculate the square root of a user-supplied number.

```
= Sqr(InputBox("Enter a number."))
```

Static Statement

Description Used at the procedure level to declare a **Static** variable and allocate storage space.

Syntax **Static** *variablename*[([*subscripts*])] [**As** *type*] [, *variablename*[([*subscripts*])] [**As** *type*]] . . .

Remarks The **Static** statement uses these arguments.

Argument	Description
variablename	Name of a variable.
subscripts	Dimensions of an array. You can declare multiple dimensions. The syntax of *subscripts* is described below.
As *type*	Reserved word used to declare the data type of the variable. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Variant**, a user-defined type, or an object data type (but no arrays of objects). Use a separate **As** *type* clause for each variable being defined.

The argument *subscripts* has this syntax:

[*lower* **To**]*upper*[, [*lower* **To**] *upper*] . . .

The **To** reserved word provides a way to indicate both the lower and upper bounds of an array variable's subscripts. The following statements are equivalent if there is no **Option Base** statement:

```
Static A(8,3)
Static A(0 To 8, 0 To 3)
Static A(8, 0 To 3)
```

Array subscripts can be negative. **To** can be used to specify any range of subscripts between –32,768 and 32,767, inclusive:

```
Static A(-4 To 10)
Static B(-99 To -5, -3 To 0)
```

The maximum number of array dimensions allowed in a **Static** statement is 60.

You can calculate the amount of memory used by a numeric array by multiplying the number of elements in the array times the number of bytes required by the data type of the array. For example, an **Integer** array, which requires 2 bytes per element, can contain twice as many elements as a **Long** array, which requires 4 bytes per element. An **Integer** array can contain four times as many elements as a **Double** array, which requires 8 bytes per element. **String** arrays are limited to less than 64K bytes because some small storage overhead is required for each array.

If you use a subscript that is greater than the specified maximum or smaller than the specified minimum, an error occurs and a message is displayed unless it is trapped in error-handling code. An error also occurs if the size of the array (in terms of bytes of memory used) exceeds the allowable limits described above.

Use **Static** in nonstatic procedures to explicitly declare **Static** variables. You must use **Static** to declare a fixed-size array in nonstatic procedures. In **Static** procedures, you can use either **Static** or **Dim** to declare **Static** variables.

You can also use a **Static** statement within a procedure to declare the data type of a **Static** variable. For example, the following statement declares a fixed-size array of integers:

```
Static EmployeeNumber(200) As Integer
```

If you try to use a **Static** statement to declare an array variable that has already been declared, an error occurs and a message is displayed.

Note The **Static** statement and the **Static** reserved word affect the lifetime of variables differently. If you declare a procedure using the **Static** reserved word (as in `Static Sub CountSales`), the storage space for all local variables within the procedure is allocated once and the value of the variables is preserved the entire time the program is running. For nonstatic procedures, storage space for variables is allocated each time the procedure is called and released when the procedure is exited. The **Static** statement is used to declare variables within nonstatic procedures to preserve their value as long as the program is running.

Variables are initialized at compile time. Numeric variables are initialized to 0, **Variant** variables to Empty. Variable-length strings are initialized as zero-length strings, and fixed-length strings are filled with zeros. The fields of user-defined type variables are initialized as if they were separate variables.

See Also **Dim** Statement, **Function** Statement, **Global** Statement, **Option Base** Statement, **ReDim** Statement, **Sub** Statement

Example This example uses the **Static** statement to declare a **Static** variable in a procedure in which the variables wouldn't usually be **Static**. Run the example several times to see that the value of the **Static** variable is preserved between calls.

```
Sub StaticDemo ()
  Static A As Integer                    ' Declare static variable.
  A = A + 1: B = B + 1                   ' Increment two variables.
  MsgBox "A = " & A & " B = " & B        ' Display results.
End Sub
```

StatusBarText Property

Applies To Controls (bound object frame, check box, combo box, command button, list box, option button, option group, subform, text box, toggle button) on a form.

Description Specifies the text that is displayed in the status bar when a control is selected.

Setting Enter a maximum of 255 characters.

Remarks You can use the StatusBarText property to provide specific information. For example, when a text box has the focus, a brief description can tell the user what kind of data to enter.

If you create a control by dragging a field from the field list, the field's Description is copied to the StatusBarText.

Access Basic Use a string expression to set the value of this property.

Design view: read/write. Other views: read-only.

See Also Caption Property

StDev, StDevP Functions

Description Return estimates of the standard deviation for a population or a population sample represented as a set of values contained in a specified field on a query, form, or report.

Syntax **StDev**(*expr*)

StDevP(*expr*)

Remarks **StDevP** evaluates a population, and **StDev** evaluates a population sample.

The **StDev** and **StDevP** functions use the following argument.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to evaluate, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

If the underlying query contains fewer than two records (or no records, for **StDevP**), these functions return a **Null** (which indicates that a standard deviation can't be calculated).

You can use **StDev** and **StDevP** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **Count** Function; **DStDev, DStDevP** Functions; **First**, **Last** Functions; **Min**, **Max** Functions; **Sum** Function; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to estimate the standard deviation of the sales amounts for orders shipped to the United Kingdom. You can enter these expressions in the SQL dialog box in the Query window.

```
SELECT StDev([Order Amount]) FROM Orders WHERE [Ship Country] = 'UK';
SELECT StDevP([Order Amount]) FROM Orders WHERE [Ship Country] = 'UK';
```

The next example estimates the standard deviation of the sales amounts for all of the underlying records in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition that limits the search to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= StDev([Order Amount])
```

Stop Statement

Description Suspends execution of the running Access Basic code.

Syntax **Stop**

Remarks You can place **Stop** statements anywhere in the program to suspend program execution.

The **Stop** statement suspends program execution, but unlike **End**, doesn't close files or clear variables. To continue a program suspended by the **Stop** statement, choose Continue or Single Step from the Run menu in the Module window.

Under certain circumstances, you also may be able to restart program execution immediately after the **Stop** statement by entering a statement in the Immediate window. For example, if the **Stop** statement occurs in an error handler, you can enter a **Resume** statement in the Immediate window to resume program execution. If the **Stop** statement occurs in a subroutine (not a **Sub** procedure), you can enter a **Return** statement in the Immediate window to continue the program.

You can continue a suspended program at a specific point by entering a **GoTo** statement with an appropriate line number or line label in the Immediate window. If you don't use

line numbers or line labels, you can specify which line you want to have executed next by using Set Next Statement from the Module window Run menu and then choosing Continue or Single Step to begin execution.

See Also **End** Statement

Example For each step through this **For...Next** loop, **Stop** suspends execution. To resume program execution, choose Continue from the Module window Run menu.

```
For I = 1 To 10                    ' Start For...Next loop.
  Debug.Print I                    ' Print I to Immediate window.
  Stop                             ' Stop each time through.
Next I
```

StopAllMacros Action

Description Stops all macros that are currently running.

Remarks You typically use this action when an error condition makes it necessary to stop all macros. You can use a conditional expression in the macro's action row that calls this action. When the expression evaluates to a certain value, Microsoft Access stops all macros.

For example, you might have a macro that displays a message box as one of a number of complex actions, including running other macros. If the user chooses Cancel in this message box, the StopAllMacros action can stop all the macros that are running.

If a macro has turned echo off, the StopAllMacros action automatically turns echo back on.

See Also Echo Action, StopMacro Action

Access Basic This action isn't available from Access Basic.

StopMacro Action

Description Stops the currently running macro.

Remarks You typically use this action when a condition makes it necessary to stop the macro. You can use a conditional expression in the macro's action row that calls this action. When the expression evaluates to a certain value, Microsoft Access stops the macro.

For example, you might create a macro that opens a form showing the daily order totals for the date entered in a pop-up form. You could use a conditional expression to be sure that the Order Date field on the pop-up form contains a valid date. If it doesn't, the MsgBox action can display an error message and the StopMacro action can stop the macro.

If a macro has turned echo off, the StopMacro action automatically turns echo back on.

See Also Echo Action, StopAllMacros Action

Access Basic This action isn't available from Access Basic.

Str, Str$ Functions

Description Return a string representation of the value of a numeric expression.

Syntax **Str**[**$**](*number*)

Remarks **Str** returns a **Variant**; **Str$** returns a **String**.

Use **Str**[**$**] to convert simple numeric values to strings. Use the **Format**[**$**] function to convert numeric values you want formatted as dates, times, or currency or in other user-defined formats.

When numbers are converted to text, a leading space is always reserved for the sign of *number*. If *number* is positive, the string returned by **Str**[**$**] contains a leading blank and the plus sign is implied. **Format**[**$**] doesn't include a leading space.

The **Val** function complements **Str**[**$**] by returning the numeric value of a string.

See Also **Format**, **Format$** Functions; **Val** Function

Example In this example, **Str** returns a string representation of the value contained in the variable `Number`. Note that if the number is positive, a space for the implied plus sign will precede the first string character.

```
= Str(Number)
```

StrComp Function

Description Returns a **Variant** that indicates the result of the comparison of two string arguments.

Syntax **StrComp(** *stringexpr1*, *stringexpr2* [, *compare*])

Remarks The **StrComp** function uses these arguments.

Argument	Description
stringexpr1	Any string expression.
stringexpr2	Any string expression.
compare	Specifies the string-comparison method. The argument *compare* must be 0, 1, 2, or the value of the CollatingOrder field returned by the **ListFields** method. If *compare* is 0, string comparison is case-sensitive. If *compare* is 1, string comparison is not case-sensitive. If *compare* is 2, the string-comparison method is **Database**, which uses the New Database Sort Order. If *compare* is omitted, **StrComp** uses the string-comparison method set by the **Option Compare** statement. However, if the module doesn't contain an **Option Compare** statement, the default method is **Binary**.

The **StrComp** function returns a **Variant** that indicates the relationship between the string expressions. Regardless of underlying data type (**Variant** or **String**), each string expression argument is converted to a **Variant** and then both arguments are compared. If *stringexpr1* is less than *stringexpr2*, –1 is returned. If *stringexpr1* equals *stringexpr2*, 0 is returned. If *stringexpr1* is greater than *stringexpr2*, 1 is returned. However, if *stringexpr1* or *stringexpr2* equals a **Null**, a **Null** is returned.

See Also **Option Compare** Statement

Example In this example, **StrComp** returns 0 to indicate that the first argument is less than the second. However, if **Option Compare Binary** has been used to override **Option Compare Database** (which Microsoft Access inserts in each new module), **StrComp** returns –1.

```
=StrComp("ABCD", "abcd")
```

String Data Type

Description There are two kinds of strings:

- Variable-length strings, which each can contain up to approximately 65,535 characters. Because some storage overhead is required, a string can't actually be 64K long.
- Fixed-length strings, which each contain a declared number of characters (up to approximately 65,535).

The type-declaration character for a **String** is $ (ANSI character 36). The codes for **String** characters range from 0 to 255. **String** characters used by all applications designed for Microsoft Windows consist of ANSI characters. The first 128 characters (0–127) correspond to the letters and symbols on a standard U.S. keyboard. These first 128 characters are the same as those defined by the ASCII character set. The second 128 characters (128–255) represent special characters, such as letters in international alphabets, accents, currency symbols, and fractions.

See Also Access Basic Data Types

String, String$ Functions

Description Return a string whose characters all have a given ANSI code or are all the first character of a string expression.

Syntax 1 **String**[**$**](*number*, *charcode*)

Syntax 2 **String**[**$**](*number*, *string*)

Remarks **String** returns a **Variant**; **String$** returns a **String**.

Use **String**[**$**] to create a string that consists of one character repeated over and over.

The **String**[**$**] function uses the following arguments.

Argument	Description
number	Numeric expression indicating the length of the returned string. It must be between 0 and approximately 65,535.
charcode	ANSI code of the character to use to build the return string. It is a numeric expression that evaluates to an **Integer** between 0 and 255, inclusive.
string	String expression whose first character is used to build the return string.

If the second argument (*charcode* or *string*) in either syntax is a **Variant**, the underlying data type determines how **String[$]** uses that argument. If the argument is a **Variant** of **VarType** 8 (**String**), **String[$]** uses the first character to create the return string. If the argument is a **Variant** of **VarType** 2–7 (any numeric type), **String[$]** uses the numeric value as the ANSI code for the character used to create the return **String**. A **Variant** of **VarType** 1 (**Null**) produces an error. To convert numeric values greater than 255 to an ANSI code between 0 and 255, **String** [$] uses the remainder after the value is divided by 256 (*argument* **Mod** 256).

See Also Appendix A, "ANSI Character Set"; **Space**, **Space$** Functions

Example

In this example, **String** returns a **Variant** consisting of 10 asterisks. The ANSI code of the asterisk character is 42.

```
= String(10, 42)
```

In this example, **String** returns a **Variant** consisting of 10 asterisks by replicating the first (and only) character provided as the second argument.

```
= String(10, "*")
```

Sub Statement

Description Declares the name, arguments, and code that form the body of a **Sub** procedure.

Syntax

[Static] [Private] Sub *subname* [(*argumentlist*)]
    [*statementblock*]
    [Exit Sub]
    [*statementblock*]
End Sub

Remarks All executable code must be in **Sub** or **Function** procedures. You can't define a **Sub** procedure inside another **Sub** or **Function** procedure.

The **Sub** statement uses these arguments.

Argument	Description
Static	Indicates that the **Sub** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Sub** procedure, even if they are used in the procedure.

Argument	Description
Private	Indicates that the **Sub** procedure is accessible only to other procedures in the module in which it exists. No other procedure in any other module has access to it.
subname	Name of the procedure. **Sub** procedure names follow the same rules that constrain the names of other variables but cannot include a type-declaration character. Because **Sub** names are recognized by all procedures in all modules, *subname* cannot be the same as any other globally recognized name in the program. Such names include the names of procedures (in Access Basic or a dynamic-link library [DLL]), **Global** variables, and **Global** constants. To avoid a name conflict, you can use the **Private** reserved word to make the **Sub** local to the module in which it appears, but access to it will not be available from any procedure in any other module. Even if you use the **Private** reserved word, *subname* cannot be the same as any other module-level variable, constant, or procedure name in the same module.
argumentlist	List of variables, representing arguments, that are passed to the **Sub** procedure when it is called. Multiple variables are separated by commas. Unless identified with the **ByVal** reserved word, arguments are passed by reference, so changing an argument's value inside the **Sub** procedure changes its value in the calling procedure.
statementblock	Any group of statements that are executed within the body of the **Sub** procedure.
Exit Sub	Causes an immediate exit from a **Sub** procedure. Program execution continues with the statement following the statement that called the **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in a **Sub** procedure.

The argument *argumentlist* has this syntax:

[**ByVal**] *variable*[( )] [**As** *type*] [, [**ByVal**]*variable*[( )] [**As** *type*]] . . .

It uses these arguments.

Argument	Description
ByVal	Indicates the argument is passed by value rather than by reference. The **ByVal** reserved word cannot be used with a variable of a user-defined type or object.
variable	Name of the variable to pass as an argument. For array variables, use the parentheses but omit the number of dimensions.
As *type*	Declares the data type of *variable*. The type may be **Integer**, **Long**, **Single**, **Double**, **Currency**, **String** (variable-length strings only), **Variant**, a user-defined type, or an object data type (but no arrays of objects). Use a separate **As** *type* clause for each argument.

Sub and **End Sub** mark the beginning and end of a **Sub** procedure.

Like a **Function** procedure, a **Sub** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** procedure, which returns a value, a **Sub** procedure can't be used in an expression.

You can call a **Sub** procedure using the procedure name followed by the argument list. See the **Call** statement for specific information on how to call **Sub** procedures.

Caution **Sub** procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow. The **Static** reserved word usually is not used with recursive **Sub** procedures.

Variables used in **Sub** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Other variables used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Caution A procedure can use a variable that is not explicitly declared in the procedure, but a name conflict can occur if anything you have defined in the Declarations section has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, a **Global** or module-level constant or variable, or an object, Access Basic assumes that your procedure is referring to that module-level name. Explicitly declare variables to avoid this kind of conflict. You can use an **Option Explicit** statement to force explicit declaration of variables.

Note You can't use **GoSub**, **GoTo**, or **Return** to enter or exit a **Sub** procedure.

See Also **Call** Statement, **Dim** Statement, **Function** Statement, **Global** Statement, **Option Explicit** Statement, **Static** Statement

Example In this example, the **Sub** statement declares the name of a **Sub** procedure that is called with two input variables and displays a result.

```
A = InputBox("Length?")                ' Ask for length of rectangle.
B = InputBox("Width?")                 ' Ask for width of rectangle.
SubDemo(A,B)                           ' Call Sub with arguments.
End                                    ' End of program.

Sub SubDemo(RLen, RWid)                ' Define a Sub with two arguments.
  Area = RLen * RWid                   ' Calculate area of rectangle.
  MsgBox "Area = " & Area              ' Display result.
End Sub                                ' End of Sub.
```

Sum Function

Description Returns the sum of a set of values contained in a specified field on a query, form, or report.

Syntax **Sum**(*expr*)

Remarks The **Sum** function uses the following argument.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to add, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

Sum totals the values in a field. For example, you can use **Sum** to determine the total cost of freight charges.

The **Sum** function treats records that contain **Null** fields as having a value of 0. The following example shows how you can calculate the sum of `Order Amount` and `Freight` fields.

```
SELECT Sum("[Order Amount] + [Freight]") FROM Orders
```

You can use **Sum** in a query expression or in a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **Count** Function; **DSum** Function; **First**, **Last** Functions; **Min**, **Max** Functions; **StDev**, **StDevP** Functions; **Var**, **VarP** Functions

Example This example uses the Orders table in the NWIND.MDB database to calculate the total sales (including freight costs) for orders shipped to the United Kingdom. You can enter this expression in the SQL dialog box in the Query window.

```
SELECT Sum([Order Amount] + [Freight]) FROM Orders
➥ WHERE [Ship Country] = 'UK';
```

The next example calculates the total sales for all of the underlying records in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition that limits the search to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= Sum([Order Amount] + [Freight])
```

Switch Function

Description Evaluates a list of expressions and returns a value or an expression associated with the first expression in the list that is **True** (–1).

Syntax **Switch**(*varexpr1*, *var1* [, *varexpr2*, *var2* . . . [, *varexpr7*, *var7*]])

Remarks The **Switch** function uses the following arguments.

Argument	Description
varexpr	Expression you want to evaluate. You can include a maximum of seven *varexpr* arguments.
var	Value or expression that is returned if the corresponding *varexpr* expression is **True**.

The **Switch** function evaluates the expressions in the *varexpr* list and returns the contents of the first *var* argument whose corresponding *varexpr* expression evaluates **True.** For example, if *varexpr1* is **True**, **Switch** returns *var1*. If *varexpr1* is **False** (0), but *varexpr2* is **True**, **Switch** returns *var2*, and so on.

The following example returns a number from 1 to 3, depending on the value in the `Ship Via` field.

```
Switch([Ship Via] = 1, "Speedy", [Ship Via] = 2, "United",
➥ [Ship Via] = 3, "Federal")
```

Switch returns a **Null** if:

- None of the *varexpr* expressions is **True**.
- The leftmost *varexpr* that is **True** has a corresponding *var* that is **Null**.

Switch evaluates all of the expressions, even though it returns only one of them. For this reason, you should watch for undesirable side effects. For example, if the evaluation of any *varexpr* results in a `Division by zero` error, an error occurs.

Access Basic doesn't check the pairing of *varexpr* and *var* arguments at design time. If the arguments aren't properly paired, a run-time error occurs.

See Also **Choose** Function, **DLookup** Function, **IIf** Function, **Select Case** Statement

Example This example assumes that you have created a table named Lang, which contains country names and native languages. **Switch** evaluates the `Ship Country` and `Ship City` fields to determine the appropriate language for a specified city. The expression is shown on multiple lines for clarity; you could also enter it on a single line.

```
= Switch([Ship City] = "Melbourne", "English",
➥ [Ship City] = "Lucerne", "German",
➥ [Ship City] = "Rome", "Italian",
➥ [Ship Country] = "CH", "French", True, "English")
```

If the city is `Melbourne`, **Switch** returns `"English"`; if it is `Lucerne`, it returns `"German"`; and so on. If the city is not one of those listed, but the country is Switzerland (`"CH"` is the abbreviation for Switzerland), it returns `"French"`. Otherwise, **Switch** returns `"English"`.

SYD Function

Description Returns the sum-of-years' digits depreciation of an asset for a specified period.

Syntax **SYD**(*cost*, *salvage*, *life*, *period*)

Remarks The **SYD** function uses the following numeric arguments.

Argument	Description
salvage	Value of the asset at the end of its useful life
life	Length of the useful life of the asset
period	Period for which asset depreciation is calculated

The arguments *life* and *period* must be expressed in the same units. For example, if *life* is given in months, *period* must also be given in months. All arguments must be positive numbers.

See Also **DDB** Function, **SLN** Function

Example Using the sum-of-years' digits method, this example returns the depreciation of an asset for a specified period given the asset's initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), the total life of the asset in years (`LifeTime`), and the period for which the depreciation is calculated (`PDepr`), also in years.

```
Const YEARMONTHS = 12                    ' Number of months in a year.
Fmt = "###,##0.00"                       ' Define money format.
InitCost = InputBox("What's the initial cost of the asset?")
SalvageVal = InputBox("What's the asset's value at the end of its life?")
MonthLife = InputBox("What's the asset's useful life in months?")
Do While MonthLife < YEARMONTHS          ' Ensure period is >= 1 year.
  MsgBox "Asset life must be a year or more."
  MonthLife = InputBox("What's the asset's useful life in months?")
Loop
```

```
LifeTime = MonthLife / YEARMONTHS          ' Convert months to years.
If LifeTime <> Int(MonthLife / YEARMONTHS) Then
  LifeTime = Int(LifeTime + 1)             ' Round up to nearest year.
End If
DepYear = CInt(InputBox("For which year do you want the depreciation?"))
Do While DepYear < 1 Or DepYear > LifeTime
  MsgBox "You must enter at least 1 but not more than " & LifeTime
  DepYear = CInt(InputBox("For what year do you want the depreciation?"))
Loop
PDepr = SYD(InitCost, SalvageVal, LifeTime, DepYear)
MsgBox "The depreciation for year " & DepYear & " is " &
➥ Format(PDepr, Fmt) & "."
```

Tab Function

Description Used with the **Print #** statement and the **Print** method to move the position at which the next character is printed.

Syntax **Tab**(*column*)

Remarks This function can be used only in Access Basic code and only with the **Print #** statement or the **Print** method.

The argument *column* is an integer expression that is the column number of the new print position.

The leftmost print position on an output line is always 1. When you use the **Print #** statement to print to files, the rightmost print position is the current width of the output file, which you can set using the **Width #** statement.

When you use **Tab** on a line with other arguments to either the **Print #** statement or the **Print** method and you place no separator (a semicolon or comma) after it, a semicolon (;) is assumed except when **Tab** appears at the end of a line.

When used with forms and files, **Tab** behaves in the following way:

- If the current print position on the current line is greater than *column*, **Tab** skips to *column* on the next output line.
- If *column* is less than 1, **Tab** moves the print position to column 1.

Tab displays the following behavior only when used with files:

- If *column* is greater than the output-line width, **Tab** calculates *printposition* = *column* **Mod** *width*
- If the calculated print position is less than the current print position, printing begins on the next line at the calculated print position. If the calculated print position is greater than the current print position, printing begins at the calculated print position on the same line.

When you use the **Tab** function with the **Print** method, the print surface is divided into uniform, fixed-width columns. The width of each column is an average of the width of all characters in the point size for the chosen font. However, there is no correlation between the number of characters printed and the number of fixed-width columns those characters occupy. For example, the uppercase letter W occupies more than one fixed-width column and the lowercase letter I occupies less. Make sure your tabular columns are wide enough to accommodate wider letters.

See Also **Print #** Statement; **Space, Space$** Functions; **Spc** Function; **Width #** Statement

Example This example uses the **Tab** function to move the print position to column 40 in an output data file.

```
Open "TESTFILE" For Output As #1          ' Create a test file.
Print #1, "This is a test of the "; Tab(40); "Tab function."
Close #1                                  ' Close the file.
```

Tan Function

Description Returns the tangent of an angle

Syntax **Tan**(*angle*)

Remarks The argument *angle* can be any valid numeric expression measured in radians.

Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite an angle divided by the length of the side adjacent to the angle.

Tan returns a **Double**. If the return value is too large, an `Overflow` error occurs.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi. Pi approximately equals 3.141593.

See Also **Atn** Function, **Cos** Function, **Sin** Function

Example This example uses **Tan** to calculate the tangent of an angle with a user-specified number of degrees.

```
Pi = 4 * Atn(1)                                    ' Calculate Pi.
Degrees = InputBox("Enter an angle in degrees.")    ' Specify angle.
Radians = Degrees * (Pi / 180)                     ' Convert to radians.
Tangent = Tan(Radians)                             ' Calculate tangent.
```

TextAlign Property

See LabelAlign, TextAlign Properties

TextHeight Method

Description Returns the height of a text string as it would be printed in the current font of a **Reports** object.

Syntax [*object.*]**TextHeight**(*strexpr*)

Remarks The **TextHeight** method uses these arguments.

Argument	Description
object	**Reports** object that determines the font size upon which the text height is based
strexpr	Text string for which the text height is determined

The height is expressed in terms of the coordinate system in effect for *object* (defined by the **Scale** method). Use **TextHeight** to determine the amount of vertical space required to display the text. The height returned includes the normal leading space above and below the string, so you can use the height to calculate and position multiple lines of text within a report. If *strexpr* contains embedded carriage returns, **TextHeight** returns the cumulative height of the lines, including the leading space above and below each line.

See Also FontName, FontSize Properties; **Scale** Method; **TextWidth** Method

Example See **Print** Method

TextWidth Method

Description Returns the width of a text string as it would be printed in the current font of a **Reports** object.

Syntax [*object*.]**TextWidth**(*strexpr*)

Remarks The **TextWidth** method uses these arguments.

Argument	Description
object	**Reports** object that determines the font size upon which the text width is based
strexpr	Text string for which the text width is determined

The width is expressed in terms of the coordinate system of *object* (as defined by the **Scale** method). Use **TextWidth** to determine the amount of horizontal space required to display the text. If *strexpr* contains embedded carriage returns, **TextWidth** returns the width of the longest line.

See Also FontName, FontSize Properties; **Scale** Method; **TextHeight** Method

Example See **Print** Method

Time, Time$ Functions

Description Return the current system time.

Syntax **Time**[$][( )]

Remarks When **Time**[$] is used anywhere except in Access Basic code in a Microsoft Access module, the empty parentheses must be included to avoid confusion with a field named Time.

Time returns a **Variant** of **VarType** 7 (Date) containing a time stored internally as the fractional part of a double-precision number.

The **Time$** function returns an eight-character string of the form *hh*:*mm*:*ss*, where *hh* is the hour (00–23), *mm* is the minute (00–59), and *ss* is the second (00–59). A 24-hour clock is used; therefore, 8:00 P.M. appears as 20:00:00.

The output of the **Time$** function is equivalent to:

```
Format$(Now,"hh:mm:ss")
```

To set the system time, use the **Time$** statement.

See Also **Time**, **Time$** Statements; **Timer** Function

Example This example returns the current value of the computer system clock.

```
TimeStamp = Time
```

Time, Time$ Statements

Description Set the system time.

Syntax **Time**[$] = *expression*

Remarks For the **Time** statement, *expression* must be a **String** formatted as a time or a **Variant** of **VarType** 7 (Date) or of **VarType** 8 (**String**). If *expression* is a **String** (or a **Variant** of **VarType** 8), times can be entered using a 12- or 24-hour clock. For example, `"2:24PM"` and `"14:24"` are both valid time arguments. If *expression* is not already a **Variant** containing a valid time, **Time** attempts to convert it using the time separators specified in the WIN.INI file. If it can't be converted to a valid time, an error occurs.

For **Time$**, *expression* must have one of the following forms, where *hh* is the hour (00–23), *mm* is the minute (00–59), and *ss* is the second (00–59).

Form	Description
hh	Sets the hour; the minute and the second default to 00.
hh:mm	Sets the hour and the minute; the second defaults to 00.
hh:mm:ss	Sets the hour, the minute, and the second.

A 24-hour clock is used, so 8:00 P.M. is entered as 20:00:00.

If you use the **Time**[$] statement to set the time with versions of MS-DOS earlier than 3.3, the change remains in effect only until you change it again or turn off your computer. Many computers have a battery-powered CMOS RAM that retains date and time information when the computer is turned off. However, to permanently change the time in the CMOS RAM on computers running earlier versions of MS-DOS, you may have to use your Setup disk or perform some equivalent action. Refer to the documentation for your particular system.

See Also **Time**, **Time$** Functions

Example In this example, the **Time** statement sets the system time to a new time entered by the user. The **Time** function provides the default time in the input box.

```
Time = InputBox("Enter new time: ", "", Time)
```

Timer Function

Description Returns the number of seconds that have elapsed since 12:00 A.M. (midnight).

Syntax **Timer**

Remarks When **Timer** is used anywhere except in Access Basic code in a Microsoft Access module, the empty parentheses must be included.

You can use the **Timer** function with the **Randomize** statement to generate a seed for the **Rnd** (random-number) function. You can also use it to time programs or parts of programs.

See Also **Randomize** Statement

Example This example uses the **Timer** function to keep track of elapsed time.

```
Start = Timer                              ' Get start time.
Do
Loop Until Timer >= Start + 5              ' Loop for 5 seconds.
```

TimeSerial Function

Description Returns the time serial for a specific hour, minute, and second.

Syntax **TimeSerial**(*hour*, *minute*, *second*)

Remarks The **TimeSerial** function uses these arguments.

Argument	Description
hour	An hour between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive, or a numeric expression
minute	A minute between 0 and 59, inclusive, or a numeric expression
second	A second between 0 and 59, inclusive, or a numeric expression

To express a specific time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should conform to the accepted range of values for the unit. These values are 0 through 23 for hours and 0 through 59 for minutes and seconds. You also can specify relative times for each argument by using a numeric expression representing the number of

hours, minutes, or seconds before or after a certain time. The following example uses expressions instead of absolute time numbers. The **TimeSerial** function returns a time for 15 minutes before (`0 - 15`) six hours before noon (`12 - 6`), or 5:45:00 A.M.

```
TimeSerial(12 - 6, 0 - 15, 0)
```

The **TimeSerial** function returns a **Variant** of **VarType** 7 (Date) containing a time that is stored internally as a double-precision fractional number between 0 and .99999. This number represents a time between 0:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

If the time specified by the three arguments, either directly or by expression, falls outside the acceptable range of times, an error occurs.

See Also **Hour** Function, **Minute** Function, **Now** Function, **Second** Function, **TimeValue** Function

Example In this example, the **TimeSerial** function returns a **Variant** of **VarType** 7 (Date) containing a time created from the integer arguments.

```
= TimeSerial(H, M, S)
```

TimeValue Function

Description Returns the time represented by a **String** argument.

Syntax **TimeValue**(*stringexpression*)

Remarks The argument *stringexpression* is a **String** representing a time from 0:00:00 (12:00:00 A.M.) through 23:59:59 (11:59:59 P.M.). You can enter valid times using a 12- or 24-hour clock. For example, `"2:24PM"` and `"14:24"` are both valid time arguments.

If *stringexpression* contains date information, **TimeValue** doesn't return it. However, if *stringexpression* includes invalid date information, an error occurs.

The **TimeValue** function returns a **Variant** of **VarType** 7 (Date) containing a time that is stored internally as a double-precision fractional number between 0 and 0.99999. This number represents a time between 0:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

See Also **Hour** Function, **Minute** Function, **Now** Function, **Second** Function, **TimeSerial** Function

Example See **Minute** function

Top Property

See Left, Top Properties

Transactions Property

Applies To **Tables**, **Dynasets**.

Description Indicates whether a **Table** or **Dynaset** supports the recording of a series of changes that can later be canceled or committed.

Setting The Transactions property settings are:

Setting	Description
–1	The **Table** or **Dynaset** supports transactions.
0	The **Table** or **Dynaset** doesn't support transactions.

Usage Read-only.

Remarks If a **Table** or **Dynaset** is based entirely on Microsoft Access tables, the value of the Transactions property is **True** (–1) and you can use transactions. Tables created by other database products, however, may not support transactions. For example, you cannot use transactions in a **Dynaset** based on a Paradox table.

Inspect the value of the Transactions property before you use the **BeginTrans** statement to make sure that the **Table** or **Dynaset** supports transactions. Using **BeginTrans** or **Rollback** on a nonsupported **Table** or **Dynaset** has no effect.

See Also **BeginTrans** Statement, **CommitTrans** Statement, **Rollback** Statement

TransferDatabase Action

Description Imports or exports data between the current Microsoft Access database and another database. You can also attach a table to the current Microsoft Access database from another database. With an attached table, you have access to the table's data while the table itself remains in the other database.

Action argument	Description
Transfer Type	The type of transfer you want to make. Select Import, Export, or Attach Table from the drop-down list in the Action Arguments section of the Macro window. The default is Import.
Database Type	The type of database to import from, export to, or attach to. You can select Microsoft Access or one of the other database types from the drop-down list. The default is Microsoft Access.
Database Name	The name of the database to import from, export to, or attach to. Include the full path name. Required argument. For types of databases, such as Paradox and dBASE, that use separate files for each table, enter the directory containing the file. Enter the file name in the Source argument (to import or attach) or the Destination argument (to export). For Btrieve databases, enter the path and file name (FILE.DDF) of the Xtrieve dictionary file for the Btrieve database. For SQL databases, enter the full ODBC connect string. To see an example of a connect string, attach an external table to Microsoft Access using the Attach Table command on the File menu. Open the table in Design view and view the table properties. The text before the first "-->" in the Description property is the connect string for this table.
Object Type	The type of object to import or export. If you select Microsoft Access in the Database Type argument, you can select Table, Query, Form, Report, Macro, or Module from the drop-down list. If you select any other type of database, or if you select Attach Table for the Transfer Type argument, this argument is ignored. The default is Table.
Source	The name of the table or Microsoft Access object that you want to import, export, or attach. For some types of databases, such as Paradox or dBASE, this is a file name. Include the file name extension (such as .DBF) in the file name. Required argument.

Action argument	Description
Destination	The name of the imported, exported, or attached table or Microsoft Access object in the destination database. For some types of databases, such as Paradox or dBASE, this is a file name. Include the file name extension (such as .DBF) in the file name. Required argument. If you select Import for the Transfer Type argument and Table for the Object Type argument, Microsoft Access creates a new table containing the data in the imported table. If you import a table or other object, Microsoft Access adds a number to the name if it conflicts with an existing name. For example, if you import Table1 and Table1 already exists, Microsoft Access renames the imported table Table2. If you export to a Microsoft Access database or another database, Microsoft Access automatically replaces any existing table or other object that has the same name.
Structure Only	Specifies whether to import or export only the structure of a Microsoft Access table without any of its data. This argument applies only if you import or export between two Microsoft Access databases. Select Yes or No. The default is No.

Remarks

You can import and export tables between Microsoft Access and other types of databases. You can import and export any Microsoft Access database object if both databases are Microsoft Access databases.

If the database you're accessing requires a password, a dialog box will appear when you run the macro. Enter the password in this dialog box.

This action is similar to choosing the Import, Export, or Attach Table command from the File menu of the Database window. You can use these commands to select a data source, such as Microsoft Access or a type of database, spreadsheet, or text file. If you select a database, a series of dialog boxes will appear in which you select the type of object to import or export (for Microsoft Access databases), the name of the object, and various other options, depending on the database you are importing from or exporting or attaching to. The arguments for the TransferDatabase action reflect the options in these dialog boxes.

For a detailed description of the rules that Microsoft Access uses to import, export, and attach tables and other objects, see the topics listed in the "See Also" section.

If you want to supply index information for an attached dBASE table, first attach the dBASE table by choosing the Attach Table command from the File menu and specify the indexes in the dialog boxes for this command. Microsoft Access stores the index

information in an .INF file. You can then delete the link to the attached table. The next time you use the TransferDatabase action to attach this dBASE table, Microsoft Access will use the index information that you have specified.

See Also TransferSpreadsheet Action, TransferText Action

Access Basic

Syntax

DoCmd TransferDatabase [*transfertype*], *databasetype*, *databasename* [, *objecttype*], ➥ *source*, *destination* [, *structureonly*]

Argument	Description
transfertype	One of the following intrinsic constants: A_IMPORT A_EXPORT A_ATTACH If you leave this argument blank, the default (A_IMPORT) is assumed.
databasetype	A string expression that is the name of one of the types of databases you can use to import, export, or attach data. In the Macro window, you can view the database types in the drop-down list for the Database Type argument of the TransferDatabase action.
databasename	A string expression that is the full name, including the path name, of the database you want to use to import, export, or attach data.
objecttype	One of the following intrinsic constants: A_TABLE A_QUERY A_FORM A_REPORT A_MACRO A_MODULE This is the type of object whose data you want to import, export, or attach. You can specify an object other than A_TABLE only if you are importing or exporting data between two Microsoft Access databases. If you leave this argument blank, the default (A_TABLE) is assumed.
source	A string expression that is the name of the object whose data you want to import, export, or attach.

Argument	Description
destination	A string expression that is the name for the imported, exported, or attached object in the destination database.
structureonly	Use the reserved word **True** (–1) to import or export only the structure of a Microsoft Access table. Use the reserved word **False** (0) to import or export the structure of the table and its data. This argument applies only if you are importing or exporting data between two Microsoft Access databases. If you leave this argument blank, the default (**False**) is assumed.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example imports the `NW Sales for April` report from the Microsoft Access database `NWSALES.MDB` into the `Corporate Sales for April` report in the current database.

```
DoCmd TransferDatabase A_IMPORT, "Microsoft Access", "C:\DBS\NWSALES.MDB",
➥ A_REPORT, "NW Sales for April", "Corporate Sales for April"
```

TransferSpreadsheet Action

Description Imports or exports data between the current Microsoft Access database and a spreadsheet file.

Action argument	Description
Transfer Type	The type of transfer you want to make. Select Import or Export from the drop-down list in the Action Arguments section of the Macro window. The default is Import.
Spreadsheet Type	The type of spreadsheet to import from or export to. You can select one of a number of spreadsheet types from the drop-down list. The default is Microsoft Excel.

Action argument	Description
Table Name	The name of the Microsoft Access table to import spreadsheet data to or export spreadsheet data from. Required argument. If you select Import for the Transfer Type argument, Microsoft Access appends the spreadsheet data to this table if the table already exists. Otherwise, Microsoft Access creates a new table containing the spreadsheet data.
File Name	The name of the spreadsheet file to import from or export to. Include the full path name. Required argument. Microsoft Access creates a new spreadsheet when you export data from Microsoft Access. If the file name is the same as the name of an existing spreadsheet, Microsoft Access replaces the existing spreadsheet.
Has Field Names	Specifies whether the first row of the spreadsheet contains the names of the fields. If you select Yes, Microsoft Access will use the names in this row as field names in the Microsoft Access table when you import the spreadsheet data. If you select No, Microsoft Access treats the first row as a normal row of data. The default is No. When you export Microsoft Access table data to a spreadsheet, the field names are inserted into the first row of the spreadsheet if you have selected Yes for this argument.
Range	The range of cells to import. Leave this argument blank to import the entire spreadsheet. You can enter the name of a range in the spreadsheet or specify the range of cells to import, such as A1:E25 or A1..E25. Microsoft Access ignores this argument when you export.

Remarks

Spreadsheet data that you append to an existing Microsoft Access table must be compatible with the table's structure. Each field in the spreadsheet must be of the same data type as the corresponding field in the table, and the fields must be in the same order (unless you set the Has Field Names argument to Yes, in which case the field names in the spreadsheet must match the field names in the table).

This action is similar to choosing the Import or Export command from the File menu of the Database window. You can use these commands to select a data source, such as Microsoft Access or a type of database, spreadsheet, or text file. If you select a spreadsheet, a series of dialog boxes will appear in which you select the name of the spreadsheet and various other options. The arguments of the TransferSpreadsheet action reflect the options in these dialog boxes.

For a detailed description of the rules that Microsoft Access uses to import and export spreadsheets, see the topics in the "See Also" section.

See Also TransferDatabase Action, TransferText Action

Access Basic

Syntax

DoCmd TransferSpreadsheet [*transfertype*] [, *spreadsheettype*], *tablename*, *filename* [, *hasfieldnames*] [, *range*]

Argument	Description
transfertype	One of the following intrinsic constants: A_IMPORT A_EXPORT If you leave this argument blank, the default (A_IMPORT) is assumed.
spreadsheettype	One of the following numeric settings: 0 Microsoft Excel 1 Lotus (WKS) 2 Lotus (WK1) 3 Lotus (WK3) **Note** You can import spreadsheet data from Microsoft Excel 3.0 and 4.0 and Lotus .WK3 files, but you cannot export Microsoft Access data to these types of files. If you leave this argument blank, the default (Microsoft Excel) is assumed.
tablename	A string expression that is the name for the Microsoft Access table you want to import spreadsheet data to or export spreadsheet data from.
filename	A string expression that is the full name, including the path name, of the spreadsheet you want to import from or export to.
hasfieldnames	Use the reserved word **True** (–1) to use the first row of the spreadsheet as field names when importing or exporting. Use the reserved word **False** (0) to treat the first row of the spreadsheet as normal data. If you leave this argument blank, the default (**False**) is assumed.
range	A string expression that is a valid range of cells or the name of a range in the spreadsheet. This argument applies only to importing. Leave this argument blank to import the entire spreadsheet.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example imports the data from the specified range of the Lotus spreadsheet NEWEMPS.WKS to the Microsoft Access Employees table. It uses the first row of the spreadsheet as field names.

```
DoCmd TransferSpreadsheet A_IMPORT, 1, "Employees", "C:\LOTUS\NEWEMPS.WKS",
➥ True, "A1:G12"
```

TransferText Action

Description Imports or exports text between the current Microsoft Access database and a text file.

Action argument	Description
Transfer Type	The type of transfer you want to make. You can import data from or export data to delimited or fixed-width text files. Select Import Delimited, Import Fixed Width, Export Delimited, or Export Fixed Width from the drop-down list in the Action Arguments section of the Macro window. The default is Import Delimited.
Specification Name	The specification name for the set of options that determines how a text file is imported or exported. Required argument for fixed-width text files. You use the Imp/Exp Setup command on the File menu to create a specification for a particular type of text file, such as a delimited text file that uses tabs to separate fields and has an MDY format for dates. Select the desired options in the dialog boxes for the Imp/Exp Setup command and save the specification. You can then enter the specification name in this argument whenever you want to import or export the same type of text file. You can import or export delimited text without entering a specification name for this argument. In this case, Microsoft Access uses the defaults from the dialog boxes of the Imp/Exp Setup command.
Table Name	The name of the Microsoft Access table to import text data to or export text data from. Required argument. If you select Import Delimited or Import Fixed Width for the Transfer Type argument, Microsoft Access appends the text data to this table if the table already exists. Otherwise, Microsoft Access creates a new table containing the text data.

Action argument	Description
File Name	The name of the text file to import from or export to. Include the full path name. Required argument. Microsoft Access creates a new text file when you export data from Microsoft Access. If the file name is the same as the name of an existing text file, Microsoft Access replaces the existing text file.
Has Field Names	Specifies whether the first row of the text file contains the names of the fields. If you select Yes, Microsoft Access uses the names in this row as field names in the Microsoft Access table when you import the text data. If you select No, Microsoft Access treats the first row as a normal row of data. The default is No. When you export Microsoft Access table data to a delimited or fixed-width text file, Microsoft Access inserts the field names of your table into the first row of the text file if you have selected Yes for this argument.

Remarks

Text data that you append to an existing Microsoft Access table must be compatible with the table's structure. Each field in the text must be of the same data type as the corresponding field in the table, and the fields must be in the same order (unless you set the Has Field Names argument to Yes, in which case the field names in the text must match the field names in the table).

This action is similar to choosing the Import or Export command from the File menu of the Database window. You can use these commands to select a data source, such as Microsoft Access or a type of database, spreadsheet, or text file. If you select a text file (delimited or fixed-width), a series of dialog boxes will appear in which you can select the name of the text file and other options. The arguments of the TransferText action reflect the options in these dialog boxes.

For a detailed description of the rules that Microsoft Access uses to import and export text files, see the topics listed in the "See Also" section. These topics also describe the various types of text files in detail.

Tip An import/export specification stores the information Microsoft Access needs to import or export a text file. You can use stored specifications to import or export text data from or to similar text files. For example, you might receive weekly sales figures in a text file from a mainframe computer. You can create and save a specification for this type of data and then use the specification whenever you add this data to your Microsoft Access database.

See Also

TransferDatabase Action, TransferSpreadsheet Action

Access Basic

Syntax

DoCmd TransferText [*transfertype*] [, *specificationname*], *tablename*, *filename* [, *hasfieldnames*]

Argument	Description

transfertype	One of the following intrinsic constants:
	`A_IMPORTDELIM`
	`A_IMPORTFIXED`
	`A_EXPORTDELIM`
	`A_EXPORTFIXED`
	If you leave this argument blank, the default (`A_IMPORTDELIM`) is assumed.

specificationname	A string expression that is the name of an import or export specification you have created and saved in the current database. This argument is required for fixed-width text files. For delimited text files, you can leave this argument blank to select the default import/export specifications.
tablename	A string expression that is the name for the Microsoft Access table you want to import text data to or export text data from.
filename	A string expression that is the full name, including the path name, of the text file you want to import from or export to.
hasfieldnames	Use the reserved word **True** (–1) to use the first row of the text file as field names when importing or exporting. Use the reserved word **False** (0) to treat the first row of the text file as normal data. If you leave this argument blank, the default (**False**) is assumed.

Remarks

You can leave an optional argument blank in the middle of the syntax, but you must include the argument's comma. If you leave a trailing argument or arguments blank, don't use a comma following the last argument you specify.

Example

This example exports the data in the Microsoft Access table `External Report` to the delimited text file `APRIL.DOC` using the specification `Standard Output`.

```
DoCmd TransferText A_EXPORTDELIM, "Standard Output", "External Report",
➥ "C:\TXTFILES\APRIL.DOC"
```

Transparent Property

Applies To Control (command button).

Description Determines whether a command button is transparent.

Setting The Transparent property settings are:

Setting	Description
Yes	The button is transparent.
No	(Default) The button is visible.

Remarks You can use this property to place a transparent button over a calculated field on a form. In Form view, you can click the button to run a macro that moves the focus to a detail section or subform showing the data used for the calculation.

You can also use this property to place transparent command buttons on a picture or map. When you click the buttons, the attached macros can display forms, list boxes, or dialog boxes.

Note To hide and disable a button, use the Visible property. To disable a button without hiding it, use the Enabled property. To hide a button only when a form or report is printed, use the DisplayWhen property.

Access Basic The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write. Other views: read-only.

See Also OnPush Property

Trim, Trim$ Functions

See **LTrim**, **LTrim$**, **RTrim**, **RTrim$**, **Trim**, **Trim$** Functions

Type Statement

Description Defines a user-defined data type containing one or more elements.

Syntax

Type *usertype*
elementname **As** *typename*
[*elementname* **As** *typename*]
. . .
End Type

Remarks The **Type** statement uses these arguments.

Argument	Description
usertype	Name of a user-defined data type. It follows standard variable-naming conventions.
elementname	Name of an element of the user-defined data type. It follows standard variable-naming conventions.
typename	One of these data types: **Integer**, **Long**, **Single, Double**, **Currency**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Variant**, or another user-defined type. The argument *typename* can't be an object data type.

The **Type** statement can be used only in the Declarations section of a module. Once you have declared a user-defined type using the **Type** statement, you can declare a variable of that type in the module's Declarations section or in any procedures in the module.

Use **Dim**, **Global**, or **Static** to declare a variable to be of a user-defined type.

Line numbers and line labels aren't allowed in **Type...End Type** blocks.

User-defined types often are used with data records because data records frequently consist of a number of related elements of different data types.

The following example shows the use of static arrays in a user-defined type.

```
Type StateData
  CityCode (1 To 100) As Integer          ' Declare a static array.
  County As String * 30
End Type
Dim Washington(1 To 100) As StateData
```

In the preceding example, `StateData` includes the `CityCode` static array, and the record `Washington` has the same structure as `StateData`.

When you declare a static array within a user-defined type, its dimensions must be declared with numeric constants rather than variables.

Example

This example uses **Type** to create a user-defined type with three elements. Then a variable is declared of the user-defined type, and data is assigned to each element.

```
' The following statements should be included in the Declarations section.
Type TestRecord                          ' Create user-defined type.
  CustNum As Long
  CustName As String * 36
  Purchase As String * 24
End Type

Dim CustEntry As TestRecord              ' Declare TestRecord variable.
CustEntry.CustNum = Val(InputBox$("Enter customer number:"))
CustEntry.CustName = InputBox$("Enter customer name:")
CustEntry.Purchase = InputBox$("Enter purchase:")
```

UBound Function

Description

Returns the largest available subscript for the indicated dimension of an array.

Syntax

UBound(*array* [, *dimension*])

Remarks

This function can be used only in Access Basic code.

The **UBound** function is used with the **LBound** function to determine the size of an array. Use the **LBound** function to find the lower limit of an array dimension.

UBound uses these arguments.

Argument	Description
array	Name of an array variable.
dimension	Integer indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second dimension, and so on. If *dimension* is omitted, 1 is assumed.

UBound returns the values listed in the table below for an array with these dimensions:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

Statement	Value returned
`UBound(A, 1)`	100
`UBound(A, 2)`	3
`UBound(A, 3)`	4

See Also **Dim** Statement, **Global** Statement, **LBound** Function, **Option Base** Statement, **ReDim** Statement, **Static** Statement

Example In this example, the **UBound** statement determines the upper bound for a two-dimensional array.

```
A = Int(9 * Rnd + 2)                          ' First dimension.
B = Int(9 * Rnd + 2)                          ' Second dimension.
ReDim Array(A, B) As Integer                  ' Set dimensions.
MsgBox "Dimension 1 = " & UBound(Array, 1)    ' First UBound.
MsgBox "Dimension 2 = " & UBound(Array, 2)    ' Second UBound.
```

UCase, UCase$ Functions

Description Return a string in which all letters of an argument have been converted to uppercase.

Syntax **UCase**[**$**](*stringexpr*)

Remarks **UCase** returns a **Variant**; **UCase$** returns a **String**.

The argument *stringexpr* can be any string expression. However, only **UCase** can accept a **Variant** of **VarType** 1 (**Null**) as *stringexpr*, in which case, a **Null** is returned.

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

See Also **LCase**, **LCase$** Functions

Example This example returns an all-uppercase version of the argument string.

```
= UCase("once upon a time")
```

Unlock Statement

See **Lock**, **Unlock** Statements

Updatable Property

Applies To **Tables**, **Dynasets**.

Description Indicates whether records in a **Table** or **Dynaset** can be updated.

Setting The Updatable property settings are:

Setting	Description
–1	The records can be updated.
0	The records cannot be updated.

Usage Read-only.

Remarks A **Table** or **Dynaset** can contain fields that can't be updated. For example, you can create a **Dynaset** in which only some fields can be changed. These fields might be fixed or contain auto-incrementing data, or the **Dynaset** might be the result of a query that combines updatable and nonupdatable **Tables**.

You can use the **ListFields** method to inspect the attributes of each field before you edit a record. If the **Table** or **Dynaset** contains any nonupdatable fields, the value of the Updatable property is **False** (0). You can, however, edit the updatable fields. So long as you edit only updatable fields, no error occurs when you use the **Update** method, even if the Updatable property is **False**.

If you are working with a **Table**, you should inspect the Updatable property setting because a table is either updatable or it is not. There is no need to check the individual fields. If you are working with a **Dynaset**, however, you should inspect the field attributes because an updatable **Dynaset** can contain nonupdatable fields.

If you edit a nonupdatable field and then use the **Update** method, an error occurs.

See Also **ListFields** Method, **Update** Method

Update Method

Description Saves the contents of the copy buffer to a specified **Table** or **Dynaset**.

Syntax *object*.**Update**

Remarks The **Update** method uses the following argument.

Argument	Description
object	Name of an open **Table** or a **Dynaset**

Use **Update** to save the current record and any changes you have made to it.

Warning Changes to the current record will be lost if:

- You use the **Edit** or **AddNew** method and then move to another record without first using **Update**.
- The argument *object* refers to a **Table** and you close the **Table** without first using **Update**.
- You close the database that contains the **Table** or **Dynaset** without first using **Update**.

To edit a record, use the **Edit** method to copy the contents of the current record to the copy buffer. If you don't use **Edit** first, an error occurs when you use **Update**.

In a multiuser environment with the **Table's** or **Dynaset's** LockEdits property set to –1, the record remains locked from the time **Edit** is used until the updating is complete.

To add a new record to a **Table** or **Dynaset**, use the **AddNew** method. You can't add records to a **Snapshot** directly; you must recreate the **Snapshot** after adding records to the underlying **Tables** and using the **Update** method. Also, you can't make changes to records in a **Snapshot**.

See Also **AddNew** Method, **CreateDynaset** Method, **CreateSnapshot** Method, **Edit** Method, **OpenTable** Method

Example This example changes the title of all sales representatives in the NWIND.MDB database and uses the **Update** method to save the changes. The underlying Employees table is automatically updated along with the **Dynaset** `MySet`.

```
Dim MyDB As Database, MySet As Dynaset, MyTable As Table
Set MyDB = CurrentDB()
Set MySet = MyDB.CreateDynaset("SELECT * FROM Employees
➥ WHERE Title = 'Sales Representative';")
Do Until MySet.EOF
  MySet.Edit                              ' Enable editing.
  MySet!Title = "Account Executive"       ' Change title.
  MySet.Update                            ' Save changes.
  MySet.MoveNext                          ' Move to next record.
Loop                                      ' End of loop.
```

Tip Using an update query to change job titles might be more efficient.

UpdateMethod Property

Applies To Control (unbound object frame [linked]).

Description Determines whether records in a form or report are updated, or refreshed, automatically or manually.

Setting The UpdateMethod property settings are:

Setting	Description
Automatic	(Default) Records are updated automatically.
Manual	Records are updated manually.

Remarks When you make changes on a form, other forms or reports that use the same data might not reflect the changes. This is especially true in a multiuser environment, in which you can change the data someone else is viewing.

Microsoft Access updates records periodically according to the interval you set in the Refresh Interval in the Options dialog box.

If you set the UpdateMethod property to Automatic, Microsoft Access updates your changes as they occur, regardless of the Refresh Interval setting. If you set UpdateMethod to Manual, Microsoft Access waits until the next interval before it performs the update.

You can update the records at any time by choosing Refresh from the Records menu in Datasheet view.

Access Basic

The property settings and their values are:

Setting	Value
Automatic	0
Manual	1

Design view: read/write. Other views: read/write.

User Function

Description

Returns the name of the current user.

Syntax

User( )

Remarks

You might use the **User** function in a procedure that keeps track of the users who modify the database. If you haven't enabled system security, the **User** function returns the default, Admin.

You can establish a secure system, or you can use Microsoft Access without restrictions. Unless you establish a secure system, Microsoft Access assumes you are using the Admin user account and gives you full permissions to all database objects.

You add user accounts to restrict the objects to which users can make changes. When you enable system security, Microsoft Access prompts users for their name and personal ID number (PIN). The **User** function returns a string that contains the name of the current user.

The following example shows how to obtain the name of the current user.

```
MsgBox("The current user is: " & User())
```

Val Function

Description

Returns the numeric value of a string of characters.

Syntax

Val(*stringexpression***)**

Remarks

The argument *stringexpression* is a sequence of characters that can be interpreted as a numeric value. The **Val** function stops reading the string at the first character that it cannot recognize as part of a number. **Val** also strips blanks, tabs, and linefeeds from the argument string. For example, the following returns the value 1615198:

```
Val("    1615 198th Street N.E.")
```

Symbols and characters often considered part of a numeric value, such as the dollar sign and commas, are not recognized by **Val** as numeric. The **Val** function does recognize the radix prefixes `&O` (for octal) and `&H` (for hexadecimal). In the code below, **Val** returns the decimal value –1 for the hexadecimal value shown:

```
Val("&HFFFF")
```

The **Val** function always returns a **Double**. If the result of the **Val** function is assigned to a variable, the **Double** returned by **Val** is forced into the data type of the variable.

Tip If the string expression you want to convert to numbers contains only numbers, consider using one of the numeric data type conversion functions (**CCur**, **CDbl**, **CInt**, **CLng**, **CSng**, or **CVar**) to perform the conversion. To convert a numeric value to a **String** or **Variant**, you can use the **Format$**, **CStr**, or **CVar** function.

See Also Data Type Conversion Functions; **Format**, **Format$** Functions; **Str**, **Str$** Functions

Example This example uses **Val** to convert the number portion of a street address string to a number.

```
StreetAddress = InputBox$("Enter your street address.")
StreetNumber = Val(StreetAddress)
```

ValidationRule, ValidationText Properties

Apply To Table fields. Controls (check box*, combo box, list box, option button*, option group, text box, toggle button*) on a form.

* Except when the control is in an option group.

Description

- ValidationRule—sets the expression that is evaluated when data in a field is added or changed.
- ValidationText—specifies the text of the message that appears if the table field or control doesn't satisfy the conditions listed in the ValidationRule setting.

Setting Enter an expression for the ValidationRule and text for the ValidationText. If the ValidationRule is blank, no data validation is performed. The maximum length for each property is 255 characters.

Remarks You can use the ValidationRule and ValidationText properties to help the user enter valid data. For example, when a record is added for a new employee, you can require that the entry in the Start Date field fall between the company's founding date and the current date. If the date entered isn't in this range, you can display the message: "Start date is incorrect."

If you create a control by dragging a field from the field list, the field's ValidationRule is copied to the control's ValidationRule.

Tip The ValidationRule expression is evaluated only when the user enters data. If a field is left unchanged, the expression isn't evaluated. If you want to require an entry in a field, set the form's BeforeUpdate property to a macro or procedure that uses the **IsNull** function to verify data entry.

Access Basic Use a string expression to set the value of each property.

Design view: read/write. Other views: read-only.

ValidationText Property

See ValidationRule, ValidationText Properties

Var, VarP Functions

Description Return estimates of the variance for a population or a population sample represented as a set of values contained in a specified field of a query, form, or report.

Syntax

Var(*expr*)

VarP(*expr*)

Remarks **VarP** evaluates a population, and **Var** evaluates a population sample.

The **Var** and **VarP** functions use the following argument.

Argument	Description
expr	String expression identifying the field that contains the numeric data you want to evaluate, or an expression that performs a calculation using the data in that field. Operands in *expr* can include the name of a table field, a control on a form, a constant, or a function (which can be either intrinsic or user-defined but not one of the other SQL aggregate or domain aggregate functions).

If the underlying query contains fewer than two records (or no records, for **VarP**), these functions return a **Null** (which indicates that a variance can't be calculated).

You can use **Var** and **VarP** in a query expression or a calculated control on a form or report (except in a page header or footer).

See Also **Avg** Function; **Count** Function; **DVar**, **DVarP** Functions; **First**, **Last** Functions; **Min**, **Max** Functions; **StDev**, **StDevP** Functions; **Sum** Function

Example This example uses the Orders table in the NWIND.MDB database to estimate the variance of freight costs. You can enter these expressions in the SQL dialog box in the Query window.

```
SELECT Var([Freight]) FROM Orders WHERE [Ship Country] = 'UK';
SELECT VarP([Freight]) FROM Orders WHERE [Ship Country] = 'UK';
```

The next example calculates the estimated variance for freight costs for all of the underlying records in a form that also uses the Orders table in the NWIND.MDB database. To apply a condition that limits the search to only some records, such as those for orders shipped to the United Kingdom, set the form's Filter property. You can enter this expression in a calculated control on the form.

```
= Var([Freight])
```

Variant Data Type

Description The **Variant** data type is the default for Access Basic: it is the data type that all variables become if they are not explicitly declared to be of some other type (using **Dim**, **Global**, or **Static**). The **Variant** data type has no type-declaration character.

The **Variant** is a unique data type that may contain numeric, string, or date data as well as the special values Empty and **Null**. You can determine how the data in a **Variant** is treated by using the **VarType** function. The following table illustrates the return value of the **VarType** function and the corresponding interpretation of the contents of the **Variant**.

Return value	Interpretation of Variant
0	Empty (uninitialized)
1	**Null** (no valid data)
2	**Integer**
3	**Long** (long integer)
4	**Single** (single-precision floating-point)
5	**Double** (double-precision floating-point)

Return value	Interpretation of Variant
6	**Currency** (scaled integer)
7	Date
8	**String**

Numeric data can be any integer or real number value from –1.797693134862315E308 to –4.94066E–324 for negative values and from 4.94066E–324 to 1.797693134862315E308 for positive values. Generally, numeric **Variant** data is maintained in its original fundamental data type within the **Variant**. For example, if you assign an **Integer** to a **Variant**, subsequent operations treat the **Variant** as if it were an **Integer**. However, if an arithmetic operation is performed on a **Variant** containing an **Integer**, a **Long**, or a **Single** and the result exceeds the normal range for the original data type, the result is promoted within the **Variant** to the next larger data type. An **Integer** is promoted to a **Long,** and a **Long** and a **Single** are promoted to a **Double**. An `Overflow` error occurs when **Variant** variables containing **Currency** and **Double** values exceed their respective ranges.

Always use the **Variant** data type when the data could contain date information, Empty, or a **Null**. You can also use the **Variant** data type in place of any fundamental data type to work with data in a more flexible way. If the contents of a **Variant** variable are digits, they may be either the string representation of the digits or their actual value, depending on the context. For example:

```
Dim MyVar As Variant
MyVar = 98052
```

In the example shown above, `MyVar` contains a numeric representation—the actual value `98052`. Arithmetic operators work as expected on **Variant** variables that contain numeric values or string data that can be interpreted as numbers. If you use the + operator to add `MyVar` to another **Variant** containing a number or to a variable of a numeric data type, the result is an arithmetic sum. However, if you use either the + operator or the **&** (string concatenation) operator to add `MyVar` to another **Variant** containing obvious string data or to a **String** variable, string concatenation occurs. In the following example, `Result` equals 98,064 because 98,052 is arithmetically added to the value `12`.

```
AnotherVar = "12"
Result = MyVar + AnotherVar
```

`MyVar` can be concatenated to another variable using either of the following operators:

```
CityStateZip = "Redmond, WA " + MyVar
CityStateZip = "Redmond, WA " & MyVar
```

The value Empty denotes a **Variant** variable that hasn't been initialized. A **Variant** containing Empty is 0 if it is used in a numeric context and a zero-length string if it is used in a string context.

Don't confuse Empty with **Null**, which indicates that the **Variant** variable intentionally contains no valid data.

See Also **VarType** Function

VarType Function

Description Returns a value that indicates how a **Variant** is stored internally by Access Basic.

Syntax **VarType**(*variant*)

Remarks The argument *variant* is a variable of the **Variant** data type. The **VarType** function returns a value that provides information about the type of data stored in the **Variant**.

What value is returned	What the Variant contains
0	Empty
1	**Null**
2	**Integer**
3	**Long**
4	**Single**
5	**Double**
6	**Currency**
7	Date
8	**String**

Note You can also determine the type of data using the **Variant** constants.

See Also Access Basic Data Types, **IsDate** Function, **IsEmpty** Function, **IsNull** Function, **IsNumeric** Function

Example This example uses **VarType** to determine whether a **Variant** argument passed to the function is either a **Single** or a **Double**. **True** is returned if the argument is one of these types of data, **False** if it is not.

```
Function IsReal(VarArg)
If VarType(VarArg) = 4 Or VarType(VarArg) = 5 Then
  IsReal = True                                  ' If Single or Double.
Else
  IsReal = False                                 ' If not.
End If
End Function
```

ViewsAllowed Property

See DefaultView, ViewsAllowed Properties

Visible Property

Applies To Forms. Form sections. Report sections. Tools and controls.

Description Hides or shows a control or section.

Setting The Visible property settings are:

Setting	Description
Yes	(Default) The object is visible.
No	The object is hidden.

Remarks To hide an object only when printing, use the DisplayWhen property.

You can use the Visible property to hide a control on a form by including it in a macro run by the OnCurrent property. For example, you can hide or show a congratulatory message next to a salesperson's monthly sales total in a sales report, depending on the sales total.

Access Basic

The property settings and their values are:

Setting	Value
Yes	–1
No	0

Design view: read/write for sections and controls; not available for forms. Other views: read/write.

See Also OnFormat Property, Transparent Property

Weekday Function

Description Returns an integer between 1 (Sunday) and 7 (Saturday) that represents the day of the week for a date argument.

Syntax **Weekday**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Hour** Function, **Minute** Function, **Month** Function, **Now** Function, **Second** Function, **Year** Function

Example This example uses **Weekday** to determine the day of the week.

```
DayNum = Weekday(Now)                    ' Get the current day of the week.
If DayNum = 1 Or DayNum = 7 Then         ' See if it's a weekend day
  Msg = "Today is a weekend day."
Else                                     '  or a weekday.
  Msg = "Today is a weekday."
End If
MsgBox Msg                               ' Display message.
```

While...Wend Statement

Description Executes a series of statements in a loop as long as a given condition is true.

Syntax

While *condition*
[*statementblock*]
Wend

Remarks The argument *condition* is a numeric or string expression that evaluates true (nonzero) or false (0 and **Null**).

If *condition* is true, all statements in *statementblock* are executed until the **Wend** statement is encountered. Control then returns to the **While** statement and *condition* is again checked. If *condition* is still true, the process is repeated. If it is not true, execution resumes with the statement following the **Wend** statement.

While...Wend loops can be nested to any level. Each **Wend** matches the most recent **While**.

Note The **Option Compare** statement affects string comparison.

Caution Do not branch into the body of a **While...Wend** loop without executing a **While** statement. Doing so may cause run-time errors or program problems that are difficult to locate.

Tip **Do...Loop** provides more structure and a more flexible way to perform looping.

See Also **Do...Loop** Statement

Example This example uses **While...Wend** to print the numbers 1 through 4.

```
A = 1
While A < 5                          ' Begin loop.
  Debug.Print A                      ' Print A.
  A = A + 1                          ' Increment A.
Wend
```

Width Property

See Height, Width Properties

Width # Statement

Description Assigns an output-line width to a file.

Syntax **Width #** *filenumber*, *width*

Remarks The **Width #** statement uses these arguments.

Argument	Description
filenumber	Number used to open the file with the **Open** statement. It can be any numeric expression that evaluates to the number of an open file. Note that the number sign (#) preceding *filenumber* is not optional.
width	Numeric expression in the range 0 to 255, inclusive, that indicates how many characters appear on a line before a new line is started. If *width* equals 0, there is no limit to the length of a line. The default value for line width is 0.

The **Width #** statement permits output-line width to be changed in files that are already open.

See Also **Open** Statement, **Print #** Statement

Example This example uses the **Width #** statement to set the output-line width for a file. A test file is created, and the same characters are written to two different line widths.

```
Open "WIDDATX.DAT" For Output As #1      ' Create test file.
For I = 0 To 9                           ' Print ASCII characters
  Print #1, Chr(48 + I);                 '  0-9 to test file
Next I                                   '  all on same line.
Print #1,                                ' Start new line.
Width #1, 5                              ' Change line width to 5.
For I = 0 To 9                           ' Print 0-9 again. This
  Print #1, Chr(48 + I);                 '  time, five characters will print
Next I                                   '  before line wraps.
Close #1                                 ' Close file.
```

Write # Statement

Description Writes data to a sequential file.

Syntax **Write #** *filenumber* [, *expressionlist*]

Remarks The **Write #** statement uses these arguments.

Argument	Description
filenumber	Number used in an **Open** statement to open a sequential file. It can be any numeric expression that evaluates to the number of an open file. Note that the number sign (#) preceding *filenumber* is not optional.
expressionlist	Comma-delimited numeric and/or string expressions to be written to the file. If you omit *expressionlist*, the **Write #** statement writes a blank line to the file.

The file identified by *filenumber* must be opened in **Output** or **Append** mode.

Unlike the **Print #** statement, the **Write #** statement inserts commas between items and quotation marks around strings as they are written to the file. You don't have to put explicit delimiters in the list. **Write #** inserts a newline character after it has written the final character in *expressionlist* to the file.

Usually, the **Write #** statement writes **Variant** data to a file in the same way it writes any other Access Basic data type. However, there are some exceptions:

- If the data being written is a **Variant** of **VarType** 0 (Empty), **Write #** writes nothing to the file for that data item. However, if multiple items are written to the file, the delimiting commas themselves are written to the file.
- If the data being written is a **Variant** of **VarType** 1 (**Null**), **Write #** writes the literal `#NULL#` to the file.
- If the data being written is a **Variant** of **VarType** 7 (Date), **Write #** writes the date to the file using the fixed date format `#yyyy-mm-dd hh:mm:ss#`. When either the date or the time component is missing or zero, **Write #** writes only the part provided to the file.

See Also **Open** Statement, **Print #** Statement

Example This example uses **Write #** to write two variables to a sequential file.

```
Open "TESTFILE" For Output As 1          ' Open file for output.
UName = InputBox("Enter your name.")     ' Get user name.
UAge = InputBox("Enter your age.")       ' Get user age.
Write #1, UName, UAge                    ' Write data to file.
Close #1                                 ' Close file.
```

Xor Operator

Description Used to perform a logical exclusion on two expressions.

Syntax *result* = *expr1* **Xor** *expr2*

Remarks The **Xor** operator is a logical exclusive or operator used to evaluate two expressions. If only one of the expressions evaluates true (nonzero), *result* is **True** (–1). If either expression is a **Null** *result* is also a **Null**. When neither expression is a **Null**, *result* is determined according to the following table.

If *expr1* is	And *expr2* is	*result* is
true (nonzero)	true	**False** (0)
true	false (0)	**True** (–1)
false	true	**True**
false	false	**False**

The **Xor** operator also performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table.

If bit in *expr1* is	And bit in *expr2* is	*result* is
0	0	0
0	1	1
1	0	1
1	1	0

Bit-wise comparisons can be performed only in Access Basic code.

See Also **And** Operator, **Eqv** Operator, **Imp** Operator, **Not** Operator, Operator Precedence, **Or** Operator

Example This example prints a message depending on the value of the variables `A`, `B`, and `C`, assuming that no variable is a **Null**. If `A` = 6, `B` = 8, and `C` = 10, both expressions evaluate **False**. Because both are **False**, the **Xor** expression is also **False**.

```
If A > B Xor B = C Then
  Debug.Print "Only one comparison expression is True, not both."
Else
  Debug.Print "Both comparison expressions are True or both are False."
End If
```

Year Function

Description Returns an integer between 100 and 9999, inclusive, that represents the year of a date argument.

Syntax **Year**(*number*)

Remarks The argument *number* is any numeric expression that can represent a date and/or time from January 1, 100 through December 31, 9999, where January 1, 1900 is 2. Numbers to the left of the decimal point in *number* represent the date; numbers to the right represent the time. Negative numbers represent dates prior to December 30, 1899.

If *number* is **Null**, this function returns a **Null**.

See Also **DatePart** Function, **Day** Function, **Hour** Function, **Minute** Function, **Month** Function, **Now** Function, **Second** Function, **Weekday** Function

Example This example uses the **Year** function to determine whether or not the current year is a leap year.

```
CurYear = Year(Now)                              ' Get current year.
If CurYear Mod 4 = 0 And CurYear Mod 100 = 0 Then
  If CurYear Mod 400 = 0 Then                    ' Evenly divisible by 400?
    LeapYear = "is a leap year."
  Else                                           ' Not evenly divisible.
    LeapYear = "is a centesimal year but not a leap year."
  End If
ElseIf CurYear Mod 4 = 0 Then
  LeapYear = "is a leap year."
Else
  LeapYear = "is not a leap year."
End If
MsgBox "The current year, " & CurYear & ", " & LeapYear
```

Part 3

Appendixes

ANSI Table

0	■	32	[space]	64	@	96	`	128	■	160	[space]	192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	TT 130	‚	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	TT 131	ƒ	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	TT 132	„	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	TT 133	…	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	TT 134	†	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	TT 135	‡	167	§	199	Ç	231	ç
8	**	40	(	72	H	104	h	TT 136	ˆ	168	¨	200	È	232	è
9	**	41	)	73	I	105	i	TT 137	‰	169	©	201	É	233	é
10	**	42	*	74	J	106	j	TT 138	Š	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	TT 139	‹	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	TT 140	Œ	172	¬	204	Ì	236	ì
13	**	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	TT 147	“	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	TT 148	”	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	TT 149	•	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	TT 150	–	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	TT 151	—	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	TT 152	˜	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	TT 153	™	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	TT 154	š	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	TT 155	›	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	TT 156	œ	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	TT 159	Ÿ	191	¿	223	ß	255	ÿ

** Values 8, 9, 10, and 13 convert to backspace, tab, line feed, and carriage return characters repectively. They have no graphical representation but, depending on the application, may effect the visual display of text.

■ Indicates that this character is not supported by Microsoft Windows.

TT Indicates that this character is available only in TrueType fonts.

Microsoft Access SQL

SQL, or Structured Query Language, is often used to query, update, and manage relational databases such as Microsoft Access. When you create a query in the Query window, behind the scenes Microsoft Access constructs the equivalent SQL statement. You can view or edit the SQL statement by choosing SQL from the View menu in the Query window. After you make changes to the SQL statement, these changes will be reflected in the Query window.

You can use SQL statements in many places in Microsoft Access where you can enter the name of a table, query, or field. For example, you can enter an SQL statement as the setting for the RecordSource property of a list box to produce a list of items. You might also use an SQL statement to set the RecordSource property of a form or report or in an Access Basic module to create a **Dynaset** or **Snapshot**.

The general format of an SQL statement includes three parts:

[*parameter-declaration;*] *manipulative-statement* [*option-declaration*];

Tables 1.2–1.4 provide information on the syntax of these elements.

For additional information about how Microsoft Access implements SQL, search Help for "SQL" or for any of the Microsoft Access SQL reserved words. The *Quick Reference Guide to SQL* (Microsoft Press®, 1989) is another source of information about SQL.

Major Differences Between Microsoft Access SQL and ANSI SQL

Microsoft Access SQL is generally ANSI-86 Level 1 compliant. The tables in this appendix list the reserved words and features directly supported by Microsoft Access SQL. Certain ANSI SQL features are not implemented in Microsoft Access SQL. Conversely, Microsoft Access includes reserved words and features not implemented in ANSI SQL.

Major Differences

- The two versions have different data types. Table 1.1 lists the equivalent Microsoft Access SQL and ANSI SQL data types.
- Different rules apply to the **BETWEEN...AND** reserved word, which has the following syntax:

  expression1 [**NOT**] **BETWEEN** *expression2* **AND** *expression3*

  In Microsoft Access SQL, *expression2* can be greater than *expression3*; in ANSI SQL, *expression2* must be equal to or less than *expression3*.

- Different wildcard characters are used with the **LIKE** reserved word.

Matching character	Microsoft Access SQL	ANSI SQL
Any single character	?	_ (underscore)
Zero or more characters	*	%

- Microsoft Access SQL is generally less restrictive. For example, it permits grouping and ordering on expressions.
- Microsoft Access SQL supports more powerful expressions. For example, you can include any Access Basic or user-defined function in a Microsoft Access SQL statement.

Enhanced Features of Microsoft Access SQL

- Joins (inner and outer)
- The **TRANSFORM** statement, which provides support for crosstab queries
- Additional aggregate functions, such as **StDev** and **VarP**
- The **PARAMETERS** reserved word for defining parameterized queries

ANSI SQL Features Not Supported in Microsoft Access SQL

- Subqueries; instead, you can nest multiple queries.
- DDL statements, such as **ALTER**, **CREATE**, and **DROP**; instead, you can use the user interface.
- DCL statements, such as **COMMIT**, **GRANT**, and **LOCK**; instead, you can use Access Basic or the user interface.
- The **UNION** reserved word.
- Distinct aggregate function references.

Microsoft Access SQL Reserved Words

The following table lists the reserved words supported by Microsoft Access SQL. In addition, Microsoft Access SQL supports all user-defined and Access Basic functions.

Table 1.0 Microsoft Access SQL Reserved Words

Reserved Words			
ALL	DISTINCT	JOIN	REAL
AS	DISTINCTROW	LEFT	RIGHT
ASC	FLOAT	LEVEL	SELECT
BINARY	FROM	LONG	SET
BIT	GROUP	LONGBINARY	SHORT
BY	HAVING	LONGTEXT	SMALLINT
BYTE	IEEEDOUBLE	ON	TABLEID
CHAR[ACTER]	IEEESINGLE	OPTION	TEXT
CURRENCY	IN	ORDER	TRANSFORM
DATABASE	INNER	OWNERACCESS	UPDATE
DATETIME	INSERT	PARAMETERS	VALUE
DELETE	INT[EGER]	PIVOT	WHERE
DESC	INTO	PROCEDURE	WITH

Equivalent SQL Data Types

The following table lists the equivalent Microsoft Access SQL and ANSI SQL data types.

Table 1.1 Equivalent SQL Data Types

Microsoft Access SQL	ANSI SQL
BINARY	Not supported
BOOLEAN	Not supported
BYTE	Not supported
CURRENCY	Not supported
DATETIME	Not supported
DOUBLE	REAL
LONG	INT[EGER]
LONGBINARY	Not supported
LONGTEXT	VARCHAR
SHORT	SMALLINT

Table 1.1 Continued

Microsoft Access SQL	ANSI SQL
SINGLE	FLOAT
TEXT	CHAR[ACTER]
Not supported	NUMBER
Not supported	NUMERIC
Not supported	DECIMAL
Not supported	DOUBLE PRECISION
Not supported	PICTURE

Microsoft Access SQL Syntax

Generally, a Microsoft Access SQL statement consists of three parts: a parameter declaration, a manipulative statement, and an option declaration.

The parameter declaration has the following syntax:

PARAMETERS *parameter-definition-list*

Table 1.2 shows the syntax of each element.

Table 1.2 Syntax of Parameter Declaration Elements

Element	Syntax
parameter-definition-list	*parameter-definition* [{, *parameter-definition*}...]
parameter-definition	*parameter-name parameter-datatype*
parameter-name	*identifier*
parameter-datatype	*access-parameter-datatype-name* | ➥ *ansi-parameter-datatype-name*
access-parameter-datatype-name	**BINARY | BOOLEAN | BYTE | CURRENCY |** ➥ **DATETIME | DOUBLE | LONG |** ➥ **LONGBINARY| LONGTEXT | SHORT |** ➥ **SINGLE | TEXT**
ansi-parameter-datatype-name	**CHAR[ACTER] | FLOAT | INT[EGER] |** ➥ **REAL | SMALLINT |VARCHAR**

The manipulative statement has the following syntax:

delete-statement | *insert-statement* | *select-statement* | *select-into-statement* |
➥ *transform-statement* | *update-statement*

The option declaration has the following syntax:

WITH OWNERACCESS OPTION

Table 1.3 shows the syntax of each element.

Table 1.3 Syntax of Manipulation Statement Elements

Element	Syntax
delete-statement	**DELETE FROM** *table-name* [**WHERE** *search-condition*]
insert-statement	**INSERT INTO** *table-name* [*column-name-list*] ➥ [*in-clause*][*select-statement*]
select-statement	**SELECT** [**ALL** | **DISTINCT** | **DISTINCTROW**] ➥ *select-list table-expression*
select-into-statement	**SELECT** {**ALL** | **DISTINCT** | **DISTINCTROW**} ➥ *select-list* **INTO** *table-name* [*in-clause*] *from-clause*
transform-statement	**TRANSFORM** *function-reference select-statement* **PIVOT** ➥ *column-name*
update-statement	**UPDATE** *table-reference-list* **SET** *set-clause-list* ➥ [**WHERE** *search-condition*]

Elements Used in Microsoft Access SQL Statements

Table 1.4 shows the elements used in Microsoft Access SQL statements and their syntax.

Table 1.4 Syntax of Microsoft Access SQL Elements

Element	Syntax
approximate-numeric-literal	*exact-numeric-literal*{E | e}*signed-integer*
between-predicate	*expression* [**NOT**] **BETWEEN** *expression* **AND** ➥ *expression*
character	{Any character in the character set except the newline indicator}
character-string-literal	'{*character*}...' | "{*character*}..."
column-alias	*identifier*
column-name	[{*table-name* | *correlation-name*}.]*identifier* | *column-alias*

Table 1.4 Continued

Element	Syntax
column-name-list	*column-name* [{, *column-name*}...]
comparison-operator	< | <= | = | > | >= | < >
comparison-predicate	*expression comparison-operator expression*
correlation-name	*identifier*
exact-numeric-literal	*signed-integer*[.*unsigned-integer*] | .*unsigned-integer*
expression	A numeric expression or a string expression
from-clause	**FROM** *table-reference-list*
table-reference-list	*table-reference* [{, *table-reference*}...][*in-clause*]
table-reference	*table-name* [**AS** *correlation-name* | *joined-table*]
table-name	*base-table-name* | *querydef-name* | *attached-table-name* | ➥ *correlation-name*
base-table-name	*identifier*
querydef-name	*identifier*
attached-table-name	*identifier*
joined-table	*table-name join-type* **JOIN** *table-name* **ON** ➥ *search-condition*
join-type	**INNER | LEFT | RIGHT**
group-by-clause	**GROUP BY** *grouping-specification-list*
grouping-specification-list	*grouping-specification* [{, *grouping-specification*}...]
grouping-specification	*expression*
having-clause	**HAVING** *search-condition*
identifier	Any valid string that may be used in an expression. If *identifier* contains one of the special characters or has the same name as an SQL reserved word, it must be enclosed in brackets.
special-character	( ) , . : ; ! = * [ ]
	For more information, search Help for "identifiers in expressions."

Table 1.4 Continued

Element	Syntax
in-clause	**IN** *database-name* [*connect-string*]
database-name	*character-string-literal* or *identifier* that refers to a fully qualified database name
connect-string	*character-string-literal*
in-predicate	*expression* [**NOT**] **IN** (*expression-list*)
expression-list	*expression* [{, *expression*}...]
like-predicate	*expression* [**NOT**] **LIKE** *pattern*
pattern	*string-expression* | *character-string-literal* If the following characters appear in *pattern*, they take on special meaning: ? matches any single character. # matches any single digit (0–9). * matches any sequence of zero or more characters. [*charlist*] matches any single character in *charlist*. [!*charlist*] matches any single character not in *charlist*. [*charrange*] matches any single character in *charrange*. [!*charrange*] matches any single character not in *charrange*.
charlist	Any sequence of one or more characters
charrange	*char1–char2*
char1, *char2*	Any single character, where *char1* has a lower ordinal value than *char2* For more information, search Help for "Like."
literal	*character-string-literal* | *numeric-literal*
null-predicate	*expression* **IS** [**NOT**] **NULL**
numeric-literal	*exact-numeric-literal* | *approximate-numeric-literal*
order-by-clause	**ORDER BY** *sort-specification-list*
sort-specification-list	*sort-specification* [{, *sort-specification*}...]
sort-specification	*expression* [**ASC** | **DESC**]
predicate	*between-predicate* | *comparison-predicate* | *in-predicate* | ➥ *like-predicate* | *null-predicate*

Table 1.4 Continued

Element	Syntax
search-condition	*boolean-term* [**OR** *search-condition*]
boolean-term	*boolean-factor* [**AND** *boolean-term*]
boolean-factor	[**NOT**] *boolean-primary*
boolean-primary	*predicate* | (*search-condition*)
select-list	* | *select-sublist* [{, *select-sublist*}...]
select-sublist	*table-name*.* | *expression* [**AS** *column-alias*]
set-clause	*column-name* = {*expression* | **Null**}
set-clause-list	*set-clause* [{, *set-clause*}...]
set-function-reference	**Count**(*) | {**Avg** | **Count** | **First** | **Last** | **Min** | **Max** | **StDev** | **StDevP** | **Sum** |**Var** | **VarP**} (*column-name*)
signed-integer	{+ | –}{*digit*}...
table-expression	*from-clause* | [*where-clause*] | [*group-by-clause*] | [*having-clause*] | [*order-by-clause*]
unsigned-integer	{*digit*}...
where-clause	**WHERE** *search-condition*

Index

Symbols

! (exclamation mark)
  for left alignment 209, 211
  in pattern matching 280
" (display literal characters) 204, 205, 209, 211
(number sign)
  as digit placeholder 203
  as wildcard character 279, 280
% (percent sign)
  as percentage placeholder 203
  in SendKeys statement 434
& (ampersand)
  as character placeholder 209
  as concatenation operator 19, 359, 360
' (single quotation mark), in Rem statement 396
() (parentheses)
  operator precedence and 359
  in SendKeys statement 434, 435
* (asterisk)
  as fill character 205, 209
  as multiplication operator 19–20, 359, 360
  as wildcard character 279, 280
+ (plus sign)
  as addition operator 20–21, 359, 360
  in SendKeys statement 434
- (hyphen), in pattern matching 280
– (subtraction operator) 22, 359, 360
, (thousand separator) 204
. (decimal placeholder) 203
/ (slash)
  as date separator 204
  as division operator 23, 359, 360
: (colon)
  in Rem statement 395
  as time separator 204
< (force lowercase) 209
< (less than operator) 78–80, 359
<= (less than or equal to operator) 78–80, 359, 360
<> (not equal to operator) 79–80, 359
= (equal sign)
  as equality operator 79–80, 359
  for user-defined function 338, 339, 340, 341, 342
> (force uppercase) 209
> (greater than operator) 78–80, 359
>= (greater than or equal to operator) 79–80, 359
? (wildcard character) 279, 280
@ (character placeholder) 209
\ (backslash)
  as division operator 23–24, 359, 360
  for literal character display 204, 205, 209
[] (brackets)
  in pattern matching 279, 280
  in SendKeys statement 434
^ (caret)
  as exponentiation operator 24, 359
  in SendKeys statement 434
{} (braces), in SendKeys statement 434
~ (tilde), in SendKeys statement 434

A

Abs function 24–25
Access Basic
  data types 25–26
  databases and 346
  date/time formatting 214–215
  executing Microsoft Access actions from 149–150
  functions available in (tables) 8
  math functions derived from 137–138
  Microsoft Access functions vs. 8
  suspending execution of 453–454
  transactions and 77, 407, 472
Access control, in Open statement 344
Access, Microsoft *See* Microsoft Access
Accessibility, of Function procedure 223
Action queries, RunSQL action and 426
Actions, properties grouped by 15
Active form, displaying all records in 441
Active objects, printing 375–377
Active window
  closing 69–70
  enlarging 301
  moving or resizing 315–316
  reducing to icon 305
  sending keystrokes from program to 433–435
ActiveControl property 26
ActiveForm property 27
ActiveReport property 27–28
AddColon property 28
addition operator (+) 20–21, 359, 360

AddMenu action
described 29
in menu bar macro 339
AddNew method
described 30–31
Update method and 487
Address labels *See* Labels
AfterUpdate property 47–48
Alias, declaring procedures with 128–130
Alignment
forcing left 209, 211
GridX/GridY properties 240
LabelAlign property 270–271
LSet statement 299–300
RSet statement 410–411
TextAlign property 270–271
AllowEditing property 31–33
AllowFilters property 33
AllowUpdating property 34
ALT key, SendKeys statement and 434
Ampersand (&)
as character placeholder 209
in concatenation operator 19
And operator 35–36, 359
Angle
arctangent of 41
cosine of 85
sine of 442
tangent of 466–467
Annuity
See also Investments
FV function 227–228
IPmt function 261–262
NPer function 326–327
Pmt function 370–372
PPmt function 373–374
PV function 385–386
Rate function 389–390
ANSI character codes
Asc function for 40
Chr/Chr$ functions for 66
Deftype statements and 134
InputBox/InputBox$ functions and 258
Like operator and 279–280
reading from file 255
String/String$ functions and 457, 458
table 503
ANSI SQL, Microsoft Access SQL vs. 507–508
Any date type 36
AppActivate statement 36–37
Appearance, properties used for (list) 10
Append mode, opening file in 344, 345
AppendChunk method 37–38
Application window, activating 36–37
Applications
beginning conversation with 115–117, 118–119, 122–124
closing DDE channels to 124–126
moving to another program from 440–441
requesting data from 115–117, 121–122
running 411
sending commands to 117–118
supplying data to 120, 122–124
ApplyFilter action 39–40
Arcs, drawing 66–68
Arctangent of angle 41
Arithmetic expressions
See also Math; Numeric expressions
Function procedures and 226
Arithmetic mean, for domain values 110–111
Arithmetic operators
described 19–24
precedence 359, 360
Arrays
Access Basic functions 8
controlling subscript range in 360
declaring as variables 140–142, 393–395
Dim statement and 140–142
Erase statement for 172–173
fixed vs. dynamic 172
functions/statements used with 3
Global statement and 233
Input # statement and 256
reallocating space for 172–173
ReDim statement and 393–395
size of 233, 274–275, 484–485
Static, declaring 450–451
user-defined, declaring 232
Arrow keys, SendKeys statement and 434
Asc function 40
Assets, depreciation of 443–444, 464–465
Asterisk (*)
as fill character 205, 209
as multiplication operator 19–20
as wildcard character 279, 280
At sign (@), as character placeholder 209
Atn function 41
AutoLabel property 28
Automatic link, verifying 285
AutoRepeat property 42
AutoResize property 43
Average, for domain values 110–111
Avg function 44

B

BackColor Property 45
backslash (\)
displaying 205
as division operator 23–24
for literal character display 204, 205, 209
BACKSPACE key, SendKeys statement and 434
BackStyle property 45–46
Basic versions, QBColor function and 386
Beep action 46
Beep statement 46–47
Beginning-of-file (BOF) property 51–52
BeginTrans statement 49–50
Between...And operator 50–51
Binary comparisons of text 361
Binary-mode files
Get statement and 228, 230
Input$ function and 255
Loc function for 295
Lock/Unlock statements and 295
Open statement and 344, 345
Put statement and 383, 384
Bitmap, in command or toggle button 370
Bit-wise comparisons 363
BOF property 51–52
Bold text 194
Bold-weight type 197–198
Bookmark property 53
Bookmarkable property 54
BorderColor property 55
BorderStyle property 55–56
BorderWidth property 56
Bound controls
OldValue property 329–330
SetValue property 438
Bound object frames
See also Object frames
AddColon/AutoLabel properties 28
Scaling property 421
BoundColumn property 57
Braces ({}), in SendKeys statement 434
Brackets ([])
in pattern matching 279, 280
in SendKeys statement 434
Branching
See also Loops
GoSub...Return statements 233–234
GoTo statement 235
On Error statement 331–333
On...GoSub/On...GoTo statements 333–334
Rem statements and 396
Branching (*continued*)
While...Wend statement 497
BREAK key, SendKeys statement and 434
Buttons
Caption property 63
MsgBox function and 317, 318–319
OnPush property 342–343
ByVal argument, in Sub statement 460

C

Calculated controls, SetValue property and 438
Calculations *See* Arithmetic operators; Math; Totals
Call statement 58
Cancel property 59
CancelEvent action
described 60–61
in OnDblClick macro 336
OnEnter/OnExit property and 337
OnMenu macro and 339
OnPush macro and 342
CanGrow/CanShrink properties 61–62
CAPS LOCK key, SendKeys statement and 434
Caption property 63
Caret (^)
as exponentiation operator 24, 359
in SendKeys statement 434
Carriage return
Line Input # statement and 282
Print method and 378
Case
converting to lowercase 275–276
converting to uppercase 485
Select Case statement 429–430
Cash flow
IRR function 262–263
MIRR function 307
CCur function 96–98
CDbl function 96–98
Character placeholders (@ and &) 209
Characters
See also Strings
ANSI code for 40, 66, 503 (table)
colors for 205
converting to lowercase 275–276
converting to uppercase 485
default comparison mode for 361
font properties 194–198
forcing case of 209, 217, 218
leftmost 276
literal, Format/Format$ functions and 204–205
moving print position for 465–466

Characters (*continued*)
number of in strings 277–278
numeric values of 489–490
Print method and 378
reading from files 255, 282
rightmost in string 404–405
in SendKeys statement 434
String/String$ functions and 457–448
Charts, Item property settings 268
ChDir statement 63–64
ChDrive statement 64
Check boxes
AutoLabel property 28
DDE function 115, 117
DDESend property 122, 124
DefaultValue property 131–132
OptionValue property 361
SpecialEffect property 447–448
Child objects
linking to master 285–286
Parent property 366
Choose function 65
Chr/Chr$ functions 66
CInt function 96–98
Circle method
described 66–68
DrawMode property 155–156
DrawStyle property 156
DrawWidth property 157
FillColor property 185
FillStyle property 186
ForeColor property 201
ScaleHeight/ScaleWidth properties 418
ScaleLeft/ScaleTop properties 419
ScaleMode property 419–420
CLEAR key, SendKeys statement and 434
CLng function 96–98
Clone method 68–69
Close action 69–70
Close method 70–71
Close statement 71–72
Colon (:)
in Rem statement 395
as time separator 204
Colors
BackColor Property 45
BorderColor property 55
DrawMode property 155–156
FillColor property 185
ForeColor property 201
formatting string for 205, 211
in Line method 283, 284
Colors (*continued*)
in Print method 377
in PSet method 381–382
QBColor function 386–387
RGB code settings 386–387
RGB function 403–404
SpecialEffect property 448
specifying 283
for Yes/No data types 219
Column property 72–73
ColumnCount property 73–74
ColumnHeads property 74–75
Columns
binding control to table field 57
hiding 75
NewRowOrCol property 321–322
showing with gridlines 442
ColumnWidths property 75–76
Combo boxes
binding column to table field 57
column headings for 74–75
Column property 72–73
column widths for 75–76
DDE function and 115, 117
DDESend property and 122, 124
designing 294
drop-down list width 294
LabelAlign property 270–271
LimitToList property 281
ListRows property 292
ListWidth property 294
OnDblClick property 335–336
RowSourceType/RowSource properties 409–410
specifying column in 72–73
specifying number of columns in 73–74
text choices for 281
TextAlign property 270–271
Command buttons
AutoRepeat property 42
Cancel property 59
Caption property 63
Default property 130–131
Enabled/Locked properties 166–167
OnPush property 342–343
Picture property 370
Transparent property 482
Command/Command$ functions 76
Command line, displaying contents of 76
Commands
menu, carrying out 151–154
sending to another application 117–118

CommitTrans statement
described 76–78
Rollback statement vs. 407
Comparing string arguments 456
Comparing text data, default mode for 361
Comparison operators
bit-wise 363
described 78–80
Like operator 279–280
precedence 359, 360
Compatibility
transactions and 407, 472
TransferDatabase action 473–476
TransferSpreadsheet action 476–479
Concatenation operator (&) 19
Conditional execution, If...Then...Else statement 246–248
Conjunctions, And operator 35–36
Const statement 81–82
Constants
functions/statements used with 6–7
using in place of values 81–82
Controlling program flow, functions/statements for (list) 3
ControlName property 82
Controls
AfterUpdate property 47–48
AutoRepeat property 42
BackColor property 45
BackStyle property 45–46
BeforeUpdate property 47–48
BorderColor property 55
BorderStyle property 55–56
BorderWidth property 56
CanGrow/CanShrink properties 61–62
Caption property 63
ControlName property 82
ControlSource property 83
Default property 130–131
default value for 131–132
Enabled property 166–167
FontName/FontSize properties 196
Format property 210–220
GoToControl action 235–236
Height property 240–241
Help for 241–242
HideDuplicates property 243
Item property 267–268
label properties 28, 270–272
Left property 276–277
LimitToList property 281
LineSlant property 284–285
Controls (*continued*)
LinkChildFields/LinkMasterFields properties 285–286
ListRows property 292
ListWidth property 294
Locked property 166–167
moving focus to 235–236
OLEClass property 330
OnDblClick property 335–336
OnEnter/OnExit properties 336–337
OnPush property 342–343
OptionValue property 361
Parent property 366
Picture property 370
position properties for 276–277
referring to active 26
Requery action 400–401
RowSourceType/RowSource properties 409–410
RunningSum property 414–415
Scaling property 421
Section property 424–425
SetValue action 437
SourceObject property 445–446
SpecialEffect property 447–448
StatusBarText property 452
Top property 276–277
Transparent property 482
UpdateMethod property 488–489
Visible property 495–496
Width property 240–241
ControlSource property
DDE function and 115, 116
DDESend function and 122, 123
described 83
Conversion
data type functions 96–98
expressions to date 95–96
functions/statements used with (list) 3–4
Coordinates
See also Horizontal (x) coordinates; Vertical (y) coordinates
customizing 420
defining for Reports object 417
setting unit of measurement for 419–420
TextHeight method 467
TextWidth method 468
Copy buffer
for editing 165
saving contents of 487–488
Copying strings, without leading or trailing spaces 300
Copying user-defined variables 299–300
CopyObject action 84–85

Cosecant functions 137, 138
Cosine functions
Access Basic equivalents 137, 138
Cos function 85
Cotangent functions 137, 138
Count function 86–87
Count property
described 87–88
Forms object and 221
Reports object and 399
Counter fields 99
Country code, format expressions and 206
CreateDynaset method
described 88–90
OpenTable method vs. 358
CreateQueryDef method 90–91
CreateSnapshot method
described 91–92
OpenTable method vs. 359
Creating files, Open statement for 343
Crosstab query, Partition function for 366, 369
CSng function 96–98
CStr function 96–98
CTRL key, SendKeys statement and 434
CurDir/CurDir$ functions 93
Currency data type
comparison operators and 79
converting numeric expressions to 96–98
declaring for variable 129
described 25, 93
Format property and 210, 215–217
in Function procedures 224, 225
Global statement and 232
setting default as 134–135
user-defined 483–484
Currency fields 99
Current user, name of 489
CurrentDB function
described 94
OpenDatabase and 346
CurrentX/CurrentY properties
described 95
Print method and 378
PSet method and 381, 382
CVar function 96–98
CVDate function 95–96

D

Data
appending to Memo or OLE Object field 37–38
assigning to variables 256–257
Data (*continued*)
creating for control 83
form/report, grouped by action 15
protecting 166–167
requesting from application 115–117
searching for 187–193
sending to another application 120
specifying source of 409–410
transferring between database programs 473–476
transferring to or from spreadsheet files 476–479
transferring to or from text files 479–481
validating 490–491
Variant data information 494–495
view options for 132–133
writing to sequential file 379–380
Data mode
in OpenForm action 347
in OpenQuery action 351–352
in OpenTable action 357, 358
Data source, properties used for (list) 11
Data types
See also specific types
Access Basic 25–26
Any 36
comparison operators and 79–80
conversion functions 96–98
Currency 25, 93
defining 134–135
Double 154
FieldSize property and 183–184
forcing declaration of 361–362
in Function procedures 224, 225
Get statement and 229
indicating for constants 81
Input # statement and 256
Integer 260–261
LinkChildFields/LinkMasterFields properties and 285–286
for ListFields method 287
for ListIndexes method 289
for ListParameters method 291
for ListTables method 292–293
Long 25, 298
in ReDim statement 394, 395
Single 25, 443
specifying for variables 140–142
SQL equivalents 509–510
in Static statement 450
String 25, 457
in Sub statement 460
user-defined 483–484
Val function and 490

Data types (*continued*)
Variant 26, 492–494
Variant data information 494–495
Database comparison mode 361
Database manipulation, Access Basic functions 8
Database objects
active, printing 375–377
for current database 94
in opening database 346–348
renaming 396–397
repainting 397–398
saving and exiting 387–388
selecting 431–432
Database programs
data transfers to or from 473–476
transactions and 407, 472
Databases
closing 70–71
copying objects between 84–85
creating QueryDefs 90–91
creating Snapshots 292–294
database objects for 94
deleting queries from 136–137
invoking QueryDef action in 179
opening 346–348
Datasheet view
opening form in 348–350
opening query in 351–352
opening table in 357–358
showing gridlines in 442
switching between Form view and 132–133
Datasheets
GoToControl action 236
Print action 375–377
printing 375–377
DataType property 99–100
Date/Date$ functions 100
Date/Date$ statements 101
Date separator
Format/Format$ functions and 204
Format Property and 213
Date/time
adding time intervals 102–103
calculating time intervals 104–106
converting expressions to 95–96, 264–265
determining day of month 111–112
displaying date serial 109
Format/Format$ functions and 202, 204, 206–208
Format property and 210, 212–215
functions/statements used for (list) 4
Hour function 244
IsDate function 264–265
Date/time (*continued*)
LastUpdated property and 273
Minute function 306
Month function 310
Now function 325
Second function 423
setting or displaying date 100–101
specifying date serial 108
specifying part of 106–107
system date 100–101, 325
system time 325, 468–469
Time/Time$ functions 468–469
Time/Time$ statements 469
Timer function 470
TimeSerial function 470–471
TimeValue function 471
Weekday function 496
Year function 501
Date/Time data type 212, 292
See also Date/time
Date/Time fields 99
DateAdd function 102–103
DateCreated property 104
DateDiff function 104–106
DatePart function 106–107
DateSerial function 108
DAvg function 110–111
Day function 111–112
dBASE databases, transfers to or from 473–476
DCount function 112–113
DDB function 114–115
DDE *See* Dynamic Data Exchange; *specific functions or statements*
DDE function 115–117
DDEExecute statement 117–118
DDEInitiate function 118–119
DDEPoke statement 120
DDERequest function 121–122
DDESend function 122–124
DDETerminate statement 124–125
DDETerminateAll statement 125–126
Debug object 126
printing text on 377–379
Decimal numbers
converting to octal 328–329
hexadecimal values for 242–243
Decimal placeholder, Format/Format$ functions and 203
DecimalPlaces property 127
Declarations section
Deftype statements in 134–135
Dim statement in 141

Declarations section (*continued*)
Option Base statement in 360
Option Compare statement in 361
Option Explicit statement in 361
Declare statement 128–130
Default Property 130–131
DefaultEditing property 31–33
Defaults
array subscript bounds 360
command button 130–131
data types 134–135, 492
directory, changing 63–64
sort order 375
text data comparison mode 361
DefaultValue property 131–132
DefaultView property 132–133
Deftype statements 134–135
DEL key, SendKeys statement and 434
Delete method 135–136
DeleteQueryDef method 136–137
Deleting current record 135–136
Deleting directories 405
Deleting files from disk 269–270
Deleting records
macros or user-defined functions with 338–339
RecordCount property and 390
Depreciation
double-declining balance method 114–115
SLN function 443–444
SYD function 464–465
Derived math functions 137–138
Description property 138
Design view
alignment specifications 240
displaying page headers and footers in 365
LastUpdated property and 272–273
opening form in 348–350
opening query in 351–352
opening report in 354–356
opening table in 357–358
selecting primary key order in 375
DFirst/DLast functions 139–140
Dialog boxes
allowing user input with 257–258
Modal property and 309
MsgBox function/MsgBox statement and 317–319
SendKeys action with 433
Digit placeholders, Format/Format$ functions and 203
Dim statement
described 140–142
ReDim statement and 394
Static statement and 451
Dimensions of arrays
Dim statement and 140–142
LBound function for 274–275
ReDim statement and 393–395
UBound function for 484–485
Dimensions of objects 240–241
Dir/Dir$ functions 142–143
Directories
changing default 63–64
changing name of 320
creating 308
current, determining 93
removing 405
Disk drives *See* Drive(s)
Disk files *See* Files
Display *See* Screen display
DisplayWhen property 143–144
Division operators
floating-point numbers (/) 23
integers (\) 23–24, 35
Mod operator 309, 359
precedence 359, 360
DLast function 139–140
DLL procedures *See* Dynamic-link library procedures
DLookup function 144–146
DMin/DMax functions 146–147
Do...Loop statement
described 148
Exit statement with 180
DoCmd statement 149–150
DoEvents function/DoEvents statement 150–151
Domains
adding values in 159–160
arithmetic mean for values in 110–111
first and last values in 139–140
functions/statements used for (list) 4
looking up values in 144–146
minimum and maximum values in 146–147
number of selected records in 112–113
standard deviations for 157–159
variance estimates for 160–162
DoMenuItem action 151–154
Double arrays
Dim statement and 141
ReDim statement and 394
Static statement and 450
Double data type
comparison operators and 79
converting to 96–98
declaring variable as 129
described 25, 154
FieldSize property and 183–184

Double data type (*continued*)
in Function procedures 224, 225
Global statement and 232
setting default as 134–135
user-defined 483–484
Val function and 490
Double-precision numbers *See* Double data type
DOWN ARROW key, SendKeys statement and 434
Drawing
circles/ellipses/arcs 66–68
coordinates for 95
lines or rectangles 283–284
DrawMode property 155–156
DrawStyle property 156
DrawWidth property
described 157
Line method and 284
PSet point size and 381
Drive(s)
changing 64
current path for 93
DStDev/DStDevP functions 157–159
DSum function 159–160
Duplicate data, hiding 243
Duplicating database objects 84–85
Duplicating recordset objects 68–69
DVar/DVarP functions 160–162
Dynamic arrays
declaring 393–395
Dim/ReDim statements and 141, 393, 394
fixed arrays vs. 172
freeing memory used by 172–173
Dynamic Data Exchange (DDE)
DDE function 115–117
DDEExecute statement 117–118
DDEInitiate function 118–119
DDEPoke statement 120
DDERequest function 121–122
DDESend function 122–124
DDETerminate statement 124–125
DDETerminateAll statement 125–126
functions/statements used for (list) 4
Microsoft Access expressions 8
Dynamic-link library (DLL) procedures
See also Library files
Any data type and 36
transferring control to 57–58
Dynaset property 162–163
Dynasets
adding new record to 30–31
creating 88–90
creating with filter 186–187
Dynasets (*continued*)
deleting current records from 135–136
GoToControl action 236
LockEdits property 297
RecordCount value for 390
saving copy buffer contents to 487–488
Sort property 444–445
Transactions property 472
Updatable property 486
Update method 487–488

E

Echo action 163–164
Edit box, InputBox$ function for 257–258
Edit method 165–166
Editing
AllowEditing property 31–33
AllowUpdating property 34
DefaultEditing property 31–33
LockEdits property 297
opening QueryDef for 353–354
properties used for (list) 11
updatability and 486
Update method and 487
Ellipse, drawing 66–68
Else *See* If...Then...Else statement
Enabled property 166–167
End Function statement 223, 225, 226
END key, SendKeys statement and 434
End statement 167–168
End Sub statement 458, 459
End-of-file (EOF) function 170
End-of-file (EOF) property 170–171
ENTER key, SendKeys statement and 434
Environ/Environ$ functions 168–169
Environment-string table 169
EOF function 170
EOF property 170–171
Equal sign (=), for user-defined functions 338, 339, 340, 341, 342
Equality operator (=) 79–80, 359
Eqv operator 171–172, 359
Erase statement 172–173
Erl function 173–174
Err function 173–174
Err statement
described 174–175
On Error statement and 332
Error/Error$ functions
described 175–176
On Error function and 332

Error handling
enabling or disabling routines 331–333
functions/statements used for (list) 4
resuming program after 402–403
returning for error code 175–176
setting code to specific value 174–175
simulating errors 176–177
status functions 173–174
Error statement 176–177
ESC key, SendKeys statement and 434
Evaluation
Eval function 177–178
IIf function and 248–249
In operator 251–252
Is operator 264
IsNull function 266
IsNumeric function 267
for Null value 266
Switch function 462–463
Events
canceling 60–61
DoEvents function/DoEvents statement 150–151
Eval function and 178
Excel, Microsoft *See* Microsoft Excel
Exclamation mark (!)
for left alignment 209, 211
in pattern matching 280
Execute method 179
Execution
See also Program execution
DDEExecute statement 117–118
DoCmd statement 149–150
grouped by action 15
Rem statements and 396
of statements based on values 429–430
Exit Function statement
described 223, 224, 226
error handling and 332
Exit statement 180
Exit Sub function, error handling and 332
Exit Sub statement 459
Exiting Microsoft Access 387–388
Exp function 181
Exponentiation functions 137–138
Exponentiation operator (^) 24, 359
Exporting data *See* Import/export actions
Expressions
And operator for 35–36
Microsoft Access–only 8
External procedures, declaring 128–130

F

F1 (Help) 242
Field names, as column headings 74–75
Field values
DFirst/DLast functions 139–140
DLookup function 144–146
First/Last functions 193
Microsoft Access expressions 8
FieldName property 181
Fields
See also Memo fields; Table fields
creating Snapshots of 286–288
default value for 131–132
linking 285–286
minimum or maximum set of values in 304
moving focus to 235–236
naming 181
PrimaryKey property 375
setting values of 437
totaling values in 461
updatability of 486
FieldSize method 182–183
FieldSize property 183–184
File attributes 184–185
File input/output
concluding 71–72
EOF function in 170
functions/statements used for (list) 5
Input # statement 256–257
Input$ function 255
Open statement 343–345
File names
changing 320
matching to pattern 142–143
File numbers, next valid unused 222
FileAttr function 184–185
Files
assigning data to variables from 256–257
BOF property 51–52
changing name of 320
closing 71–72
closing all 401
controlling access to 295–296
current position in 294–295, 425
deleting from disk 269–270
EOF function 170
EOF property 170–171
file mode information on 184–185
moving to another directory 320
next numbers for 222

Files (*continued*)
opening 343–345
output line width for 498
printing to 379–380
reading characters from 255
reading into variables from 228–230, 282
saving output to 313–314
setting position in 425, 428
writing buffer data to 401
writing data to 379–380, 499
writing from variables to 382–384
FillColor property 185
Filter property 186–187
Filtering
AllowFilters property 33
ApplyFilter action 39–40
CreateDynaset method and 89
in CreateSnapshot method 92
Filter property 186–187
in OpenForm action 346, 350
in OpenReport action 354, 355, 356
removing applied filters 441
Financial operations
Access Basic functions 8
DDB function 114–115
functions/statements used for 5
FV function 227–228
IPmt function 261–262
IRR function 262–263
MIRR function 307
NPer function 326–327
NPV function 327–328
Pmt function 370–372
PPmt function 373–374
PV function 385–386
Rate function 389–390
SLN function 443–444
SYD function 464–465
Find dialog box, FindRecord action and 191
Find methods, NoMatch property and 323–324
FindFirst/FindLast/FindNext/FindPrevious methods 187–188, 323–324
FindNext action 189
FindRecord action 190–193
First function 193
Fix function 260
Fixed arrays, dynamic arrays vs. 172
Fixed-length strings 457
Focus
enabling for control 166–167
moving, macro or user-defined function with 334–335
Focus (*continued*)
moving to specified field or control 235–236
moving to specified page 237–238
referring to control with 26
referring to form with 27
referring to report with 27–28
retaining in open form 309–310
Font properties 194–198
FontBold property 194
FontItalic property 195
FontName property 196
Fonts
choosing 273–274
Print method and 377, 378
screen vs. printer 273
FontSize property 196
FontUnderline property 195
FontWeight property 197–198
Footers, page 365
For...Next loop, Exit statement with 180
For...Next statement 198–199
ForceNewPage property 200
ForeColor property 201
Form datasheets
GoToControl action 236
setting field values on 437
Form property 201–202
Form sections
CanGrow/CanShrink properties 61–62
DisplayWhen property 143–144
ForceNewPage property 200
KeepTogether property 268–269
NewRowOrCol property 321–322
SpecialEffect property 447–448
Form view
DisplayWhen property and 143–144
Enabled property and 166–167
Locked property and 166–167
opening form in 348–350
running macros or user-defined functions in 335–336
switching between Datasheet view and 132–133
Form window, AutoResize property 43
Format/Format$ functions 202–209
Format property
Date/Time data types 212–215
described 210–220
Number and Currency data types 210, 215–217
Text and Memo data types 217–218
Yes/No data types 219–220
FormatCount property 220

Formatting
currency 210, 215–217
date/time 202, 204, 206–208, 212–215
macro or user-defined function before 337–338
numbers 202–206, 210, 211–212, 215–217
scientific notation 204, 215, 216
strings 202, 209
FormName property 221
Forms
See also Form sections; Reports
AfterUpdate property 47–48
AllowEditing property 31–33
AllowFilters property 33
AllowUpdating property 34
ApplyFilter action 39–40
AutoResize property 43
BeforeUpdate property 47–48
Bookmark property 53
Caption property 63
Count function for 86–87
Count property 87–88
data grouped by action 15
default command buttons for 130–131
DefaultEditing property 31–33
DefaultView property 132–133
design properties (list) 11–12
dimensions of 240–241
Dynaset property 162–163
First function 193
Form property 201–202
FormName property 221
GoToControl action 236
GridX/GridY properties 240
Help files for 241–242
hWnd property 245
Last function 193
LayoutForPrint property 273–274
linking to embedded objects 285–286
making specified record current 238–240
menu bar customizing 29
Min/Max functions 304
modal 349
Modal property 309–310
moving to specified page on 237–238
OnClose property 340–341
OnCurrent property 334–335
OnDelete property 338–339
OnInsert property 338–339
OnMenu property 339–340
OnOpen property 340–341
OpenForm action 348–350
page numbers for 364
Forms (*continued*)
PopUp property 372
position properties 276–277
Print action 375–377
RecordLocks property 391–392
RecordSelectors property 392
RecordSource property 393
referring to active 27
ScrollBars property 422–423
Section property 424–425
setting field values on 437
SetValue action 437
ShowAllRecords action 441
ShowGrid property 442
Sum function 461
ViewsAllowed property 132–133
Visible property 495–496
Forms object
described 221–222
Count property 87–88
indicating number open 87–88
referring to 5
FreeFile function 222
Function keys, SendKeys statement and 434
Function procedures
constants as local to 81
in Declare statement 128–130
Dim statement in 141
Exit statement in 180
name, arguments, and code for 223–226
RunCode action for 412
setting default data type for 134–135
Sub procedure vs. 225
Function statement 223–226
Functions
grouped by programming task 3–8
names of 223–224
user-defined *See* User-defined functions
FV function 227–228

G

Get statement 228–230
GetChunk method 230–231
Global constants
defining 81–82
Function procedures and 226
Global statement
described 232–233
ReDim statement and 394

Global variables
declaring 232–233
Function procedures and 226
GoSub...Return statements 233–234
GoTo statement 235
GoToControl action
described 235–236
OnEnter/OnExit properties and 337
GoToPage action 237–238
GoToRecord action 238–240
Graphics
Circle method 66–68
coordinate system 95
DrawMode property 155–156
DrawStyle property 156
DrawWidth property 157
FillColor property 185
FillStyle property 186
ForeColor property 201
functions/statements used for 5
Line method 283–284
properties used for (list) 12
Graphs
DisplayWhen property 143–144
linking to forms or report 285–286
OLE object descriptions 330
Partition function for 366
Scaling property 421
Greater than operator (>) 78–80, 359
Greater than or equal to operator, (>=) 79–80, 359
Gridlines, showing in Datasheet view 442
GridX/GridY properties 240

H

Headers, page 365
Height property 240–241
Help, properties used for 12
HELP key, SendKeys statement and 434
HelpContextID/HelpFile properties 241–242
Hex/Hex$ functions 242–243
HideDuplicates property 243
Hiding controls or sections 495–496
HOME key, SendKeys statement and 434
Horizontal (x) coordinates
CurrentX property 95
Left property 276–277
ScaleLeft property 419
ScaleWidth property 418
Hour function 244
Hourglass action 244–245
hWnd property 245
hyphen (-), in pattern matching 280

I

Icons
enlarging to active window 301
reducing active window to 305
If...Then...Else statement 246–248
IIf (immediate if) function 248–249
Immediate window, sending output to 126
Imp operator 250–251, 359
Import/export actions
listed 15
TransferDatabase 473–476
TransferSpreadsheet 476–479
TransferText 479–481
In operator 251–252
Index1...Index5 properties 253
Index property 252–253
Indexed property 254
Indexed table, locating records in 426–427
Indexing
creating Snapshots 288–290
multiple indexes 254
PrimaryKey property for 375
setting current index 252–253
setting multiple-field index 253
setting single-field index 254
Inequality operator (<>) 79–80, 359
Input # statement 256–257
Input/Input$ functions 255
Input mode
Input$ function and 255
opening file in 344, 345
Input/output *See* File input/output; *specific devices*
InputBox/InputBox$ functions
described 257–258
Eval function with 178
INS (Insert) key, SendKeys statement and 434
Inserting records, macros or user-defined functions with 338–339
Inspection of variables, functions/statements used for (list) 4
InStr function 259
Int function 260
Integer arrays
Dim statement and 141
ReDim statement and 394
Static statement and 450

Integer data type
comparison operators and 79
converting to 96–98, 260
declaring variable as 129
described 25, 260–261
FieldSize property and 183–184
in Function procedures 224, 225
Global statement and 232, 233
for page numbers 364
setting default as 134–135
user-defined 483–484
Interest calculation
IPmt function 261–262
IRR function 262–263
Internal rate of return
IRR function 262–263
MIRR function 307
Interest rate per period, Rate function 389–390
Investments
annuity payments 370–372
future value of 227–228
interest payment on 261–262
interest rate per period 389–390
net present value of 327–328
number of periods for 326–327
present value of 385–386
principal payments 373–374
I/O statements, in Function procedures 226
IPmt function 261–262
IRR function 262–263
Is operator 264
IsDate function 96, 264–265
IsEmpty function 265
IsNull function 266
IsNumeric function 267
Italics, FontItalic property 195
Item property 267–268

J

Justifying string variables
left 299–300
right 410–411

K

KeepTogether property 268–269
Keystrokes
sending to active window 433–435
sending to program 432–435
Kill statement 269–270

L

Labels
AddColon/AutoLabel properties 28
BackColor property 45
Caption property 63
ForeColor property 201
FormatCount property 220
LabelAlign property 270–271
LabelX/LabelY properties 271–272
placing for new control 271–272
TextAlign property 270–271
SpecialEffect property 447–448
Last function 193
LastUpdated property 272–273
Layout
See also Measurements
LayoutForPrint property 273–274
MoveLayout property 313–314
NewRowOrCol property 321–322
LBound function 274–275
LCase/LCase$ functions 275–276
Leading spaces, copying strings without 300
LEFT ARROW key, SendKeys statement and 434
Left/Left$ functions 276
Left property 276–277
Left-aligning string variables 299–300
Len function 277–278
Less than operator (<) 78–80, 359
Less than or equal to operator (<=) 78–80, 359
Let statement 277–278
Letter range, in defining data type 134–135
Letters *See* Characters
Library files, declaring procedures from, 128–130
See also Dynamic-link library (DLL) procedures
Like operator 279–280, 359, 360
LimitToList property 281
Line Input # statement 282
Line method
described 283–284
DrawMode property and 155–156
DrawStyle property and 156
DrawWidth property and 157
FillColor property and 185
FillStyle property and 186
ForeColor property and 201
ScaleHeight property and 418
ScaleLeft property and 419
ScaleMode property and 419–420
ScaleTop property and 419
ScaleWidth property and 418

Lines
drawing *See* Line method
LineSlant property 284–285
output, width for 498
Width property 240–241
LineSlant property 284–285
LinkChildFields/LinkMasterFields properties 285–286
Linked object frames
data description 267–268
SourceObject property 445–446
UpdateMethod property 488–489
List boxes
binding column to table field 57
column headings for 74–75
column widths for 75–76
DefaultValue property 131–132
OnDblClick property 335–336
RowSourceType/RowSource properties 409–410
specifying column in 72–73
specifying number of columns in 73–74
specifying width of 294
ListFields method 286–288
ListIndexes method
described 288–290
Seek method and 426
ListParameters method 290–291
ListRows property 292
ListTables method 292–294
ListWidth property 294
Literal characters, formatting and displaying 204–205, 209
Loans
annuity payments 370–372
future value of 227–228
interest payment on 261–262
interest rate per period 389–390
number of periods for 326–327
present value of 385–386
principal payments 373–374
Loc function 294–295
Local area networks *See* Multiuser environment
Locating records 187–188
Lock Read/Lock Write, in Open statement 345
Lock statement 295–296
Locked property 166–167
Locked records 391–392
LockEdits property 297
LOF function 297–298
Log function 298
Logarithmic functions 138, 181, 298, 442
Logical operators
And 35–36
Eqv 171–172
Imp 250–251
Not 324–325
Or 363–364
precedence 359, 360
Xor 359, 500
Long arrays
Dim statement and 141
ReDim statement and 394
Static statement and 450
Long data type
comparison operators and 79
converting to 96–98
declaring variable as 129
described 25, 298
FieldSize property and 183–184
in Function procedures 224, 225
Global statement and 232
in ListFields method 287
in ListIndexes method 289
in ListParameters method 291
in ListTables method 293
in Partition function 367
for RGB color value 403–404
in Right/Right$ function 404
setting default as 134–135
user-defined 483–484
Long integer data type *See* Long data type
Lookups, DLookup function 144–146
Loops
Do...Loop statement 148
Exit statement 180
For...Next statement 198–199
GoTo statements and 235
While...Wend statement and 497
Lotus 1-2-3, data transfers to or from 476–479
Lowercase
converting characters to 275–276
forcing characters to 209, 217, 218
LSet statement 299–300
LTrim/LTrim$ functions 300

M

Macros
AfterUpdate property and 47–48
AutoRepeat property 42
before formatting 337–338
before printing or previewing 341–342
BeforeUpdate property and 47–48

Macros (*continued*)
CancelEvent action and 60–61
carrying out commands from 151–154
on closing form or report 340–341
on command button choice 342–343
CurrentX/CurrentY properties for 95
on deleting records 338–339
FindRecord action in 191
hourglass pointer for 244–245
on inserting records 338–339
menu bar 339–340
on moving focus 334–335
on opening form or report 340–341
preventing stoppage of 438–439
properties used for (list) 12
on receiving or losing focus 336–337
running 412–414
running OnDblClick 335–336
stopping all 454
stopping current 454–455
updating screen display during 163–164
Margins
left and top 276–277
right and bottom 240–241
Master objects, linking child objects to 285–286
Math
arithmetic operators 19–24
derived functions 137–138
exponentiation functions 137–138
functions/statements used for (list) 5
logarithmic functions 138, 181, 298, 442
Mod operator 309
operator precedence 359–360
Sum function 461–462
Max function 304
Maximize action 301
Measurements
alignment grid 240
custom coordinate system 420
Height property 240–241
Left property 276–277
ListWidth property 294
Scale method 417
ScaleHeight/ScaleWidth properties 418
ScaleLeft/ScaleTop properties 419
ScaleMode property 419–420
setting unit of 165
Top property 276–277
twips 277
Width property 240–241
Memo, appending data to 37–38
Memo fields
described 99
GetChunk method 230–231
number of bytes in 182–183
from specified recordset 230–231
Memo formats 217–218
Menus
adding commands to 151–154
customizing 29, 339–340
DoMenuItem action 151–154
Message box, displaying 316–317
Methods
grouped by programming task 7
recordset objects grouped by 9
Microsoft Access, Access Basic functions vs. 8
Microsoft Basic Compiler, QBColor function and 386
Microsoft Excel
data transfers to or from 476–479
DDE functions with 115–126
Item property and 267, 268
running 411
Microsoft Paintbrush, for picture buttons 370
Microsoft PowerPoint, running 411
Microsoft QuickBasic, QBColor function and 386
Microsoft Windows
calls from Access Basic to 245
DDE functions and 115–126
font choices and 196
Help applications 242
running application in 411
sending keystrokes to 432–433
SendKeys statements and 435
Microsoft Word for Windows
for creating Help file 242
running 411
Mid/Mid$ functions 302
Mid/Mid$ statements 303
Min function 304
Minimize action 305
Minus sign (–), as subtraction operator 22
Minute function 306
MIRR function 307
MkDir statement 308
Mod operator
described 309
Spc function and 447
Modal properties, setting 349
Modal property 309–310
Modal vs. modeless forms 349
Module level, declaring variables at 140–142
Monitor *See* Screen display
Month, determining day of 111–112

Month function 310
Mortgages *See* Loans
Mouse, OnDblClick property 335–336
Mouse pointer, changing to hourglass 244–245
MoveFirst/MoveLast/MoveNext/MovePrevious methods 311–312
MoveLayout property 313–314
MoveSize action 315–316
Moving active window 315–316
Moving among records 311–312
Moving files, Name statement for 320
Moving to next printing location 313–314
MS-DOS applications, running 411
MS-DOS version, Lock/Unlock statements and 296
MsgBox, Eval function results in 178
MsgBox action 316–317
MsgBox function/MsgBox statement 317–319
Multiplication operator (*) 19–20, 359, 360
Multiuser environment
  LastUpdated property and 273
  Lock/Unlock statements in 295–296
  Open statement in 344–345
  RecordLocks property 391–392
  User function 489

N

Name of current user 489
Name statement 320
Naming controls 82
Naming database objects 396–397
Naming fields 181
Naming forms or reports 221
Navigation buttons
  ScrollBars property and 422
Negation, Not operator 324–325, 359
Net present value (NPV) function 327–328
Network environment *See* Multiuser environment
New page, forcing 200
NewRowOrCol property 321–322
NextRecord property 313–314
NoMatch property 323–324
Not equal to operator (<>) 79–80, 359
Not operator 324–325, 359
Now function 325
NPer function 326–327
NPV function 327–328
Null field, inspecting for 249
Null reserved word, Is operator and 264
Null string, Format/Format$ functions and 203
NUM LOCK key, SendKeys statement and 434
Number data types *See* Numeric expressions
Number fields
  described 99
  maximum data size for 183–184
Number sign (#)
  as digit placeholder 203
  as wildcard character 279, 280
Numeric arrays, ReDim statement and 394
Numeric expressions
  See also Arithmetic expressions; Math
  absolute value for 24–25
  assigning to variables 277–278
  Atn function 41
  Avg function 44
  Between...And operator 50–51
  comparison operators 78–80
  converting data types, 96-98 *See also specific types*
  converting to dates 95–96
  converting to integers 260
  converting to strings 202–203, 455
  converting Variant variables to 267
  Cos function 85
  CurrentX/CurrentY properties 95
  DecimalPlaces property 127
  division remainder of 309
  Format/Format$ functions 202–206
  Format property 210, 211–212, 215–217
  Global statement 233
  hexadecimal values for 242–243
  logarithms of 138, 181, 298
  negating 324–325
  Oct/Oct$ functions 328–329
  in On...GoSub/On...GoTo statements 333
  random-number generator 388
  reading from sequential files 256–257
  Sgn function 439
  Sin function 442
  Sqr function 449
  standard deviation for 157–159, 452–453
  Tan function 466–467

O

Object frames
  linking OLE objects to 267–268
  OLE object descriptions 330
  Scaling property 421
  UpdateMethod property 488–489
Object manipulation
  functions/statements used for (list) 5
  grouped by action 15

Object references
assigning to variables 278–279
properties used for (list) 12–13
Objects
See also Object manipulation; Object references; *specific types of objects*
active, printing 375–377
assigning to variables 436
closing 69–70
copying between databases 84–85
Debug 126
dimensions of 240–241
displaying 143–144
Dynaset, creating 88–90
Forms 221–222
hiding 143–144, 495–496
If...Then...Else statements and 248
linking to forms or reports 285–286
Parent property for 366
position properties for 240–241, 276–277
Reports 399
Screen 422
specifying data source for 409–410
specifying position of 276–277
transferring between database programs 473–476
Oct/Oct$ (octal value) functions 328–329
Offset number, GoToRecord action and 238, 239
OldValue property 329–330
OLE object
described 330
linking to object frames 267–268
methods used with 7
properties used for 13
specifying number of columns for 73–74
OLE object fields
appending data to 37–38
described 99
number of bytes in 182–183
from specified recordset 230–231
OLEClass property 330
On...GoSub/On...GoTo statements 333–334
On Error Resume Next errors 175
On Error statement 331–333
OnClose property 340–341
OnCurrent property 334–335
OnDblClick property 335–336
OnDelete property 338–339
OnEnter property 336–337
OnExit property 336–337
OnFormat property
described 337–338
FormatCount property and 220
OnFormat property (*continued*)
MoveLayout/NextRecord/PrintSection properties and 313–314
OnInsert property 338–339
OnMenu property 339–340
OnOpen property 340–341
OnPrint property
described 341–342
PrintCount property and 380–381
OnPush property
described 342–343
Eval function and 178
Open files
controlling access to 295–296
current position in 294–295
file handle or file mode of 184–185
Name statement and 320
size of 297–298
Open statement
described 343–345
LOF function and 297
Put statement and 383, 384
OpenDatabase function 346–348
OpenForm action 348–350
Opening files 343–345
Opening forms
described 348–350
macros or user-defined functions with 340–341
OpenQuery action 351–352
OpenQueryDef method 353–354
OpenReport action 354–356
OpenTable action 357–358
OpenTable method 358–359
Operating environment
returning string for 168–169
yielding event processing to 150–151
Operator precedence 359–360
Operators
arithmetic 19–24
comparison 78–80, 279–280
functions/statements used with (list) 6
logical *See* Logical operators
Option Base statement
described 360
ReDim statement vs. 394
Option buttons
AutoLabel property 28
DefaultValue property 131–132
OnDblClick property 335–336
OptionValue property 361
SpecialEffect property 447–448
Option Compare statement 361

Option Explicit statement
described 361–362
Function procedures and 226
variable name conflict and 460
Option groups
ControlSource property 83
DDE function and 115, 116
DDESend property and 122, 123
DefaultValue property 131–132
OptionValue property and 361
OptionValue property 361
Or operator 359, 363–364
Ordering records, Index property for 252–253
Output line width, assigning 498
Output mode, opening file in 344, 345

P

Page
ForceNewPage property 200
KeepTogether property 268–269
moving to first field on 237–238
Page breaks
Top property 276–277
PAGE DOWN key, SendKeys statement and 434
Page numbers 364
Page property 364
PAGE UP key, SendKeys statement and 434
PageHeader/PageFooter properties 365
Paintbrush, Microsoft *See* Microsoft Paintbrush
Paradox databases
transactions and 407, 472
transfers to or from 473–476
Parameters, QueryDef, creating Snapshot from 290–291
Parent property 366
Parentheses ()
operator precedence and 359
in SendKeys statement 434, 435
Partition function 366–369
Path, determining for specified drive 93
Pattern-matching features 279–280
Payments
IRR function 262–263
MIRR function 307
Pmt function 370–372
PPmt function 373–374
PV function 385–386
Rate function 389–390
Percent sign (%), in SendKeys statement 434
Percentage placeholder (%), Format/Format$ functions and 203
Picture property 370
Plus sign (+)
as addition operator 20–21
in SendKeys statement 434
Pmt function 370–372
PopUp property
described 372
OpenForm action and 349
Position
in current file 294–295, 425
custom coordinate system 420
Left property 276–277
in Line method 283
LineSlant property 284–285
properties used for 13
Tab function and 465–466
Top property 276–277
PowerPoint, Microsoft *See* Microsoft PowerPoint
Preserve reserved word, ReDim statement and 393, 395
Previewing
See also Print Preview
macros or user-defined functions before 337–338, 341–342
MoveLayout/NextRecord/PrintSection properties 313–314
ScaleHeight/ScaleWidth properties and 418
ScaleLeft/ScaleTop properties and 419–420
ScaleMode property and 419–420
PrimaryKey property 375
Principal payment, PPmt function 373–374
Print # statement
described 379–380
skipping spaces in 446–447
Tab function and 465–466
Write # statement vs. 499
Print action 375–377
Print method
described 377–379
font styles for 194
FontItalic/FontUnderline properties and 195
FontName/FontSize properties and 196
ForeColor property and 201
ScaleHeight/ScaleWidth properties and 418
ScaleLeft/ScaleTop properties and 419
ScaleMode property and 419–420
skipping spaces in 446–447
Tab function and 465–466
Print Preview
opening form in 348–350
opening query in 351–352
opening report in 354–356

Print Preview (*continued*)
opening table in 357–358
RecordLocks property and 391–392
PRINT SCREEN key, SendKeys statement and 434
Print Setup, NewRowOrCol property and 321–322
PrintCount property 380–381
Printer fonts, screen fonts vs. 273–274
Printing
Access Basic functions 8
DisplayWhen property and 143–144
to file 379–380
ForeColor property and 201
graphics, functions/statements used for 5
horizontal and vertical coordinates for 95
KeepTogether property 268–269
LayoutForPrint property 273–274
macros or user-defined functions before 337–338, 341–342
MoveLayout property 313–314
NewRowOrCol property 321–322
NextRecord property 313–314
OnPrint property 341–342
OpenReport action for 355, 356
page headers and footers 365
Print action 375–377
Print method 377–379
PrintCount property 380–381
PrintSection property 313–314
properties used for 13
RecordLocks property and 391–392
ScaleHeight/ScaleWidth properties and 418
ScaleLeft/ScaleTop properties and 419–420
ScaleMode property and 419–420
to screen vs. printer 273–274
specifying page number 364
PrintSection property 313–314
Private reserved word
in Function statement 223
in Sub statement 459
Procedures
branching to specified lines in 235
declaring variables in 140–142, 450–451
ending Access Basic 167–168
external, declaring 128–130
functions/statements used in (list) 6
ReDim statement and 393, 395
subroutine handling in 233–234
Program control, transferring to Sub or DLL procedure 57–58
Program execution
See also Execution
Access Basic function for 8
Program execution (*continued*)
Exit Function statements and 224
grouped by action 15
On Error statement and 331, 332
resuming after error 402–403
Shell function and 440–441
suspending 453–454
terminating 167–168
Programming tasks, properties grouped by 10–14
Programs
See also Program execution
data transfers between 473–481
remarks in 396
Properties
CancelEvent action and 60–61
grouped by action 15
grouped by programming task 10–14
recordset objects grouped by 10
setting values of 437
PSet method
described 381–382
DrawMode property and 155–156
DrawWidth property and 157
ScaleHeight/ScaleWidth properties and 418
ScaleLeft/ScaleTop properties and 419
ScaleMode property and 419–420
Put statement 382–384
PV function 385–386

Q

QBColor function
described 386–387
Line method and 283
PSet method and 381
Queries
ApplyFilter action 39–40
Between...And operator 50–51
calculating number of records in 86–87
creating new QueryDef 90–91
deleting from database 136–137
Execute method 179
First/Last functions 193
Min/Max functions 304
OpenQuery action 351–352
Partition function 366–369
RecordSource property 393
RunSQL action 416–417
ShowAllRecords action 441
showing all records in 441
SQL property 448–449

Queries (*continued*)
Sum function 461
Switch function 462–463
totaling values on 461
Query dynaset
GoToControl action 236
making specified record current 238–240
QueryDef objects
closing 70–71
creating Snapshots from 91–92, 290–291, 292–294
deleting from database 136–137
invoking 179
opening for editing 353–354
SQL property 448–449
Question mark (?), as wildcard character 279, 280
Quit action 387–388
Quotation mark ('), in Rem statement 396
Quotation marks ("), for literal character display 204, 205, 209

R

Randomize statement
described 388
Timer function and 470
Random-mode files
Get statement and 228, 229
Loc function for 294–295
Lock/Unlock statements and 295
Open statement for 344, 345
Put statement and 383
Random-number generator
initializing 388
Rnd function 405–406
Rate function 389–390
Read access 344
Read operations
access for 344
setting file position for 428
Read Write access 344
Read-only data, Locked property for 166–167
Receipts
IRR function 262–263
MIRR function 307
Record manipulation, methods used for 7
Record position
BOF property 51–52
EOF property 170–171
methods used for (list) 7
RecordCount property 390–391
RecordLocks property 391–392
Records
adding to Table or Dynaset 30–31
adding values in 159–160
calculating number of 86–87
copying for editing 165–166
creating Snapshots with 286–291, 292–294
current
moving records to 311–312
setting bookmark for 52–53
specifications for 187–188
specifying 238–240
deleting, macros or user-defined functions with 338–339
deleting current 135–136
Dynaset property 162–163
editing 165–166
filtering 33, 39
finding
Find actions 189, 190–193
Find methods 187–188
first and last, values in 139–140, 193
Index property for 252–253
inserting, macros or user-defined functions with 338–339
locating
with Find methods 187–188
in indexed table 426–427
locking and unlocking 295–296
looking up field value in 144–146
methods used with 7
Microsoft Access expressions 8
moving focus among, macros or user-defined functions with 334–335
moving position of 311–312
number of in recordset 390–391
number selected in domain 112–113
ordering, Index property for 252–253
position of *See* Record position
properties used with 13
RecordCount property 390–391
restricting at printing 354–356
searching repeatedly for 189
selecting from table 186–187
set of *See* Domains
standard deviations for 157–159
Updatable property for 486
UpdateMethod property settings 488–489
updating 487
variance estimates for 160–162
Records menu
AllowEditing command 31–33
Refresh command 398, 488

RecordSelectors property 392
Recordset objects
closing 70–71
duplicating 68–69
grouped by method 9
grouped by property 10
Recordsets
BOF property 51–52
Bookmark property 53
Bookmarkable property 54
changing *See* Transactions
creating Snapshots from 286–288
EOF property 170–171
Memo or OLE Object fields from 230–231
methods used with 7
moving among records in 311–312
NoMatch property 323–324
properties used with 13
RecordCount property 390–391
RecordSource property 393
Rectangles
BackColor property 45
BackStyle property 45–46
BorderWidth property 56
drawing 283–284
SpecialEffect property 447–448
Recursive procedures
Function procedures 225
Sub procedures 460
subroutines 234
ReDim statement
described 393–395
dynamic arrays and 141
Global variables and 233
Refresh command (Records menu)
RepaintObject action vs. 398
UpdateMethod property and 488
Refresh interval 488
Rem statement 396
Removing directory 405
Rename command (MS-DOS), Name statement vs. 320
Renaming database object 82
Renaming file or directory 320
RepaintObject action 397–398
Repeating keys, in SendKeys statement 435
Repeating statements
Do...Loop 148
For...Next 198–199
Report property 201–202
Report sections
CanGrow/CanShrink properties 61–62
ForceNewPage property 200
Report sections (*continued*)
KeepTogether property 268–269
NewRowOrCol 321–322
OnFormat property 337–338
OnPrint property 341–342
SpecialEffect property 447–448
Visible property 495–496
Reports
See also Forms; Report sections
Count function 86–87
CurrentX/CurrentY properties 95
design properties (list) 11–12
dimensions of 240–241
DrawMode property 155–156
DrawStyle property 156
DrawWidth property 157
FillColor property 185
FillStyle property 186
First function 193
FontBold property 194
FontItalic/FontUnderline properties 195
FontName/FontSize properties 196
ForeColor property and 201
FormatCount property 220
FormName property 221
graphics functions/statements (list) 5
GridX/GridY properties 240
Help files for 241–242
hWnd property 245
Last function 193
LayoutForPrint property 273–274
linking to embedded objects 285–286
Min/Max functions 304
OnOpen/OnClose properties 340–341
OpenReport action 354–356
Page property 364
Print action 375–377
PrintCount property 380–381
RecordLocks property 391–392
RecordSource property 393
referring to 399
referring to active 27–28
Report property 201–202
ScaleHeight/ScaleWidth properties 418
ScaleLeft/ScaleTop properties 419
ScaleMode property 419–420
Section property 424–425
setting field values on 437
SetValue action 437
Sum function 461
Visible property 495–496

Reports object
circle, ellipse, or arc on 66–68
Count property 87–88
defining coordinate system for 417
described 399
indicating number of open 87–88
lines or rectangles on 283–284
printing text on 377–379
setting point to specified color on 381–382
text height and 467
text width and 468
Requery action
described 400–401
RepaintObject action vs. 398
Reserved words, SQL 509
Reset statement 401
Resizing active window 315–316
Restore action 402
Resume action 396–397
Resume statement 402–403
Return statement 234–235
RGB color code, QBColor function 386–387
RGB function
described 403–404
Line method and 283, 284
PSet method and 381
RIGHT ARROW key, SendKeys statement and 434
Right/Right$ functions 404–405
Right-aligning string variables 410–411
RmDir statement 405
Rnd function 405–406
Rollback statement 406–408
Rows
maximum number (combo box) 292
NewRowOrCol property 321–322
showing with gridlines 442
RowSourceType/RowSource properties 409–410
RSet statement 410–411
RTrim/RTrim$ functions 300
RunApp action 411
RunCode action 412
RunMacro action 412–414
RunningSum property 414–415
RunSQL action 416–417

S

Saving copy buffer contents 487–488
Saving current record 487
Saving transaction changes 407
Savings *See* Investments
Scale method 417
ScaleHeight property 418
ScaleLeft property 419
ScaleMode property 419–420
ScaleTop property 419
ScaleWidth property 418
Scaling property 421
Scientific notation 204, 215, 216
Screen display
DisplayWhen property and 143–144
enlarging window on 301
properties used for (list) 11
removing window from 305
SpecialEffect property and 447–448
updating 397–398
updating during macro 163–164
Visible property and 495
Screen fonts, printer fonts vs. 273
Screen object 422
SCROLL LOCK key, SendKeys statement and 434
ScrollBars property 422–423
Searching
Find actions 189, 190–193
Find methods 187–188
InStr function 259
Move methods 311–312
Seek method 426
Secant functions 137, 138
Second function 423
Section property 424–425
Sections *See* Form sections; Report sections
Seek function 425
Seek method
described 426–427
NoMatch property and 323–324
Seek statement 428
Select Case statement
described 429–430
If...Then...Else statements vs. 248
On...GoSub/On...GoTo statements vs. 333
Select queries, Partition function for 366, 368–369
SelectObject action 431–432
SendKeys action 432–433
SendKeys statement 433–435
Sequential files
assigning data to variables from 256–257
Lock/Unlock statements and 295
opening 344, 345
printing to 379–380
reading characters from 255
writing data to 379–380, 499
Set of records *See* Domains
Set statement 436

SetValue action
described 437
RepaintObject action and 398
SetWarnings action 438–439
Sgn function 439
Shared lock type 345
SHARE.EXE, Lock/Unlock statements and 296
Shell function 440–441
SHIFT key, SendKeys statement and 434
ShowAllRecords action 441
ShowGrid property 442
Showing controls or sections 495–496
Sign of number 439
Sin (sine) function 137, 138, 442
Single data type
comparison operators and 79
converting to 96–98
declaring variable as 129
described 25, 443
FieldSize property and 183–184
in Function procedures 224, 225
Global statement and 232
setting default data type as 134–135
user-defined 483–484
Single-precision numbers *See* Single data type
Size
See also Dimensions of arrays; Dimensions of objects
of file, determining 297–298
properties used for 13
slash (/), as division operator 23
Snapshots
CreateSnapshot method 91–92
creating with filter 186–187
ListFields method 286–288
ListIndexes method 288–290
ListParameters method 290–291
ListTables method 292–294
RecordCount value for 390–391
Sort property 444–445
Sort order
See also Sorting
for Like operator 280
primary key and 375
Sort Property 444–445
Sorting
See also Sort order
Apply Filter action 39–40
Index1...Index5 properties 253
Index property 252–253
Indexed property 254
Sort Property 444–445
Sound, Beep action/Beep statement for 46–47
SourceObject property 445–446
Space/Space$ functions 446
Spaces
strings consisting of 446
trimming from strings 300
Spc function 446–447
Speaker, generating sounds through 46–47
SpecialEffect property 447–448
Spreadsheets, data transfers to or from 476–479
SQL operations
See also SQL WHERE clause
ApplyFilter action 39–40
creating Snapshot objects 91–92
data type equivalents 509–510
described 507–514
functions/statements used in 6, 511–514
Microsoft Access vs. ANSI 507–508
properties used in 14
RecordSource property and 393
RunSQL action 416–417
SQL property 448–449
syntax 510–511
SQL statements 507, 510, 511
SQL WHERE clause
ApplyFilter action 39–40
Filter property 186–187
in OpenForm action 346–347, 350
in OpenReport action 355, 356
Sqr function 449
Square root function 449
Standard deviation functions 157–159, 452–453
Starting Microsoft Access, Command/Command$ functions 76
Statements
in Function procedures 224
grouped by programming task 3–8
repeating 198–199
repeating block of 148
Static arrays
declaring 450–451
in user-defined data types 483–484
Static attribute, in Function statement 223
Static reserved word
described 451
Function procedures and 225
in Sub statement 458
Static statement 450–451
Statistics, standard deviation functions 157–159
StatusBarText property 452
StDev/StDevP functions 452–453
Stop statement 453–454

StopAllMacros action 454
StopMacro action 455
Storage space
allocating 140–142, 232–233
allocating at procedure level 450
allocating for Static variable 450–451
freeing 172
Len function and 277–278
reallocating 172–173, 393–395
Str/Str$ functions 455
Straight-line depreciation (SLN function) 114–115
StrComp function 456
String arrays
Dim statement and 141
ReDim statement and 394
Static statement and 450
String characters *See* Characters
String concatenation operator (&) 19, 359, 360
String data type
See also Data
ANSI code for 40, 66
Command$ function 76
comparison operators and 79, 80
converting to and from numbers 96–98
converting to dates 95–96
declaring for variable 129
described 25, 457
in Function procedures 224, 225
Global statement 232
Input$ function 255
in ListFields method 287
in ListIndexes method 289
in ListParameters method 291
in ListTables method 292
setting default as 134–135
Space$ function 446
user-defined 483–484
String/String$ functions 457–458
Strings
See also Text
ANSI code specifications 457, 458
assigning to variables 278–279
colors for 205
comparing 279–280, 456
concatenation operator (&) 19, 359, 360
converting numbers to 202–203, 455
converting to lowercase 275–276
converting to uppercase 485
copying without leading or trailing spaces 300
evaluating 177–178
fixed-length 457
formatting 202, 205, 209
Strings (*continued*)
functions/statements used with (list) 6
Global statement and 233
height of 467
justifying variables in 299–300
left-aligning 299–300
leftmost characters in 276
number of characters in 277–278
numeric value of 489–490
reading from files 256–257
reading lines from file into 282
replacing parts of 303
returning parts of 302
right-aligning 410–411
rightmost characters in 404–405
searching for first occurrence of 259
of spaces 446
specifications for 457–458
time represented by 471
uppercase 485
variable-length 457
width of 468
Sub procedures
Access Basic, transferring control to 57–58
constants as local to 81
in Declare statements 128–130
Dim statement in 141
ending 167–168
error handling and 331
Exit statement in 180
Function procedure vs. 225
names for 459
RunCode action with 412
Sub statement and 458–461
variables in 460
Sub statement 458–461
Subdirectories, creating 308
Subforms
CanGrow/CanShrink properties 61–62
linking to form or report 285–286
OnEnter/OnExit properties 336–337
Parent property 366
SourceObject property 445–446
Subreports
linking to form or report 285–286
Parent property 366
SourceObject property 445–446
Subroutines, GoSub...Return statements 233–234
Subtraction operator (–) 22, 359, 360
Sum function 461–462
Sum of values, in specific domain 159–160
Sum operator (+) 20–21

Sum-of-years' digits depreciation method 464–465
Switch function 462–463
SYD function 464–465
System date 100–101, 325
System messages, turning on or off 438–439
System time 325, 468–469

T

Tab function 465–466
TAB key, SendKeys statement and 434
Table datasheets, GoToControl action 236
Table design, properties used in 14
Table fields
  binding controls to 83
  Caption property 63
  CreateDynaset method 88–90, 358
  DataType property 99–100
  default values for 131–132
  Description property 138
  FieldName property 181
  first and last, values from 139–140
  Format property 210–220
  Indexed property 254
  looking up values in 144–146
  maximum data size for 183–184
  PrimaryKey property 375
  protecting data in 166–167
  validating data entry in 490–491
Table objects
  creating for editing 358–359
  DateCreated property 104
  Index property 252–253
  LastUpdated property 272–273
  LockEdits property 297
  Transactions property 472
  Updatable property 486
  Update method 487–488
Tables
  See also Table objects; Table fields
  adding new record to 30–31
  binding combo or list box column to 57
  CreateDynaset method and 358
  creating Snapshots from 91–92, 288–290, 292–294
  deleting current record from 135–136
  GoToRecord action 238–240
  Index1...Index5 properties 253
  locating indexed records in 426–427
  locating records in (Find methods) 187–188
  making specified record current 238–240
  OpenTable action 357–358
Tables (*continued*)
  OpenTable method 358–359
  PrimaryKey property 375
  RecordSource property 393
  saving copy buffer contents to 487–488
  selecting records from 186–187
  ShowAllRecords action 441
  TransferDatabase action and 473–476
  TransferSpreadsheet action and 476–479
  TransferText action and 479–481
Tan (tangent) functions 137, 138, 466–467
Text
  See also Strings
  default comparison mode for 361
  Description property 138
  display properties (list) 14
  FontItalic/FontUnderline properties 195
  FontName/FontSize properties 196
  FontWeight property 197–198
  formatting 210, 211, 217–218
  height of 467
  remarks 396
  status bar 452
  transferring to or from text file 479–481
  for validating entries 490–491
  width of 468
Text boxes
  AddColon/AutoLabel properties 28
  CanGrow/CanShrink properties 61–62
  DDE function and 115, 116
  DDESend property and 122, 123
  DecimalPlaces property 127
  Format property 210–220
  LabelAlign property 270–271
  RunningSum property 414–415
  ScrollBars property 422–423
  TextAlign property 270–271
  ValidationRule/ValidationText properties 490–491
Text comparisons 361
Text display, properties used in 14
Text fields
  described 99
  maximum data size for 183–184
Text files, data transfers to or from 479–481
Text formats 210, 211, 217–218
TextAlign property 270–271
TextHeight method 467
TextWidth method 468
Then *See* If...Then...Else statement
Thousand separator, Format/Format$ functions and 204
Tilde (~), in SendKeys statement 434
Time *See* Date/time

Time intervals
adding to date 102–103
between two specified dates 104–106
in Partition function 367
specifying part of 106–107
Time separator, Format/Format$ functions and 204
Time/Time$ functions 468–469
Time/Time$ statements 469
TimeSerial function 470–471
TimeValue function 471
Toggle buttons
DefaultValue property 131–132
Enabled/Locked properties 166–167
OnDblClick property 335–336
OptionValue property 361
Picture property 370
Tool bar, custom 370
Tools
AddColon/AutoLabel properties 28
BackColor property 45
BackStyle property 45–46
BorderColor property 55
BorderStyle property 55–56
BorderWidth property 56
Column property 72–73
Height property 240–241
Help files for 241–242
LabelAlign property 270–271
LabelX/LabelY properties 271–272
ListRows property 292
ListWidth property 294
ScrollBars property 422–423
SpecialEffect property 447–448
TextAlign property 270–271
Visible property 495–496
Width property 240–241
Top property 276–277
Totals
DSum function 159–160
RunningSum property 414–415
Sum function 461–462
Trailing spaces, copying strings without 300
Transactions
beginning 49–50
defined 49
reversing and ending 406–408
saving and ending 76–78
Transactions property 472
TransferDatabase action 473–476
TransferSpreadsheet Action 476–479
TransferText action 479–481
Transparent border 55–56
Transparent circle or line 186
Transparent controls
BackStyle property 45–46
Transparent property 482
Trim/Trim$ functions 300
TrueType font 273
Twips, defined 277
Type statement
described 483–484
ending 167–168
Type-declaration characters, with constant name 81

U

UBound function
described 484–485
LBound function and 274
UCase/UCase$ functions 485
Unbound object frames
See also Object frames
Item property 267–268
LinkChildFields/LinkMasterFields 285–286
OLEClass property 330
RowSourceType/RowSource properties 409–410
Scaling property 421
SourceObject property 445–446
UpdateMethod property 488–489
Underlining, FontUnderline property 195
Undoing transaction changes 407
Unlock statement 295–296
UP ARROW key, SendKeys statement and 434
Updatable property 486
Update method 487–488
UpdateMethod property 488–489
Updating
AllowUpdating property 34
BeforeUpdate/AfterUpdate properties 47–48
RecordLocks property and 391–392
Refresh interval for 488
RepaintObject action 397–398
Requery action 400–401
Updatable property 486
Update method 487–488
UpdateMethod property 488–489
Uppercase, forcing characters to 209, 217, 218
User function 489
User input
in dialog box 257–258
validating 490–491
User instructions, macro for displaying 336

User-defined data types
See also Variant data type
described 26, 483–484
in Function procedures 225
Get statement and 229, 230
User-defined errors, simulating occurrence of 176–177
User-defined formats 210–220
User-defined functions, running
before formatting 337–338
before printing or previewing 341–342
BeforeUpdate/AfterUpdate properties and 47–48
on command button choice 342–343
on double-clicking 335–336
on inserting or deleting 338–339
on moving focus 334–335
on opening or closing form or report 340–341
on receiving or losing focus 336–337
User-defined variables
Global statement and 233
Let statement and 279
LSet statement and 299–300

V

Val function 489–490
ValidationRule/ValidationText properties 490–491
Values
arithmetic mean for set of 110–111
assigning to variables 277–278
comparing to list 251–252
in first or last records (domain) 139–140
in first or last records (query, form, or report) 193
in Function procedures 224, 225
maximum and minimum 146–147, 304
returning for expressions 177–178
selecting from list 65
setting for field, control, or property 437
standard deviation for 157–159, 452–453
statement execution based on 429–430
of strings 489–490
Sum function 461–462
table field defaults 131–132
twips 277
unedited bound control 329–330
using constants for 81–82
variance estimates for 160–162, 491–492
Var function 491–492
Variable-length strings 457
Variables
assigning object reference to 436
assigning sequential file data to 256–257
Variables (*continued*)
assigning unedited bound control value to 329–330
assigning values to 278–279
checking for initialization 265
Variables (*continued*)
copying 299–300
data types for 140–142
declaring 140–142, 393–395, 450–451
declaring global 232–233
forcing declaration of 361–362
in Function procedures 224, 225, 226
functions/statements used with (lists) 4, 6–7
operating environment 168–169
reading from file into 228–229, 282
setting default data type for 134–135
Static 450–451
storage space required for 277–278
in Sub procedures 460
writing to disk file from 382–384
Variance functions
DVar/DVarP 160–162
Var/VarP 491–492
Variant data type
Command function and 76
comparison operators and 79, 80
constants vs. 81
converting string expressions to 96–98
converting to date 95–96
converting to numeric expression 267
declaring for variable 129
described 26, 492–494
in Function procedures 224, 225
Global statement and 232, 233
Input function and 255
IsDate function and 264–265
IsEmpty function and 265
IsNull function and 266
Print # statement and 380
Put statement and 383
Right function and 404
setting default as 134–135
StrComp function and 456
String function and 446
time intervals with 102–103, 104–106
TimeSerial function and 471
type of data stored in 494–495
user-defined 483–484
Write # statement and 499
VarP function 491–492
VarType function 494–495

Vertical (y) coordinates
CurrentY property 95
ScaleHeight property 418
ScaleTop property 419
Top property 276–277
ViewsAllowed property 132–133
Visible property 495–496

W

Warnings, system message 438–439
Weekday function 496
Where condition *See* SQL WHERE clause
While...Wend statement 497
Width property 240–241
Width # statement 498
Wildcard characters
Kill statement and 269
Like operator and 279–280
Window mode, in OpenForm action 347, 349, 350
Windows
See also Form window
active, sending keystrokes to 433–435
closing 69–70
enlarging 301
Modal property and 309
moving 315–316
properties used for 14
reducing to icon 305
resizing 301, 315–316
restoring to previous size 402
Windows-based applications *See* Microsoft Windows
Word for Windows, Microsoft *See* Microsoft Word for Windows
Worksheets, Item property settings 268
Write access 344
Write operations
access for 344
in closing all files 401
to sequential files 379–380, 499
setting file position for 428
from variables to files 382–384
Write # statement 499

X

Xor operator 359, 500

Y

Year function 501
Yes/No data types, formats for 219–220
Yes/No fields 99