The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbCalGreg	0	Indicates that the Gregorian calendar is used.
vbCalHijri	1	Indicates that the Hijri calendar is used.

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbBlack	0x0	Black
vbRed	0xFF	Red
vbGreen	0xFF00	Green
vbYellow	0xFFFF	Yellow
vbBlue	0xFF0000	Blue
vbMagenta	0xFF00FF	Magenta
vbCyan	0xFFFF00	Cyan
vbWhite	0xFFFFFF	White

Comparison Constants

The following constants are defined in the Visual Basic for Applications type library and can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbUseCompareOption	-1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	For Microsoft Access (Windows only), performs a comparison based on information contained in your database.

Compiler Constants

Visual Basic for Applications defines constants for exclusive use with the **#If...Then...#Else** directive. These constants are functionally equivalent to constants defined with the **#If...Then...#Else** directive except that they are global in scope; that is, they apply everywhere in a project.

On 16-bit development platforms, the compiler constants are defined as follows:

Constant	Value	Description
Win16	**True**	Indicates development environment is 16-bit.
Win32	**False**	Indicates that the development environment is not 32-bit.

On 32-bit development platforms, the compiler constants are defined as follows:

Constant	Value	Description

Vba6	**True**	Indicates that the development environment is Visual Basic for Applications, version 6.0.
Vba6	**False**	Indicates that the development environment is not Visual Basic for Applications, version 6.0.
Win16	**False**	Indicates that the development environment is not 16-bit.
Win32	**True**	Indicates that the development environment is 32-bit.
Mac	**False**	Indicates that the development environment is not Macintosh.
Win16	**False**	Indicates that the development environment is not 16-bit.
Win32	**False**	Indicates that the development environment is 32-bit Windows.
Mac	**True**	Indicates that the development environment is Macintosh.

Note These constants are provided by Visual Basic, so you cannot define your own constants with these same names at any level.

Date

Constants

The following constants can be used anywhere in your code in place of the actual values:

Argument Values

The *firstdayofweek* argument has the following values:

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *firstdayofyear* argument has the following values:

Constant	Value	Description

vbUseSystem	0	Use NLS API setting.
VbUseSystem DayOfWeek	0	Use the day of the week specified in your system settings for the first day of the week.
VbFirstJan1	1	Start with week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full week of the year.

Return Values

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Date Format Constants

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. For real numbers, display a data and time. If there is no fractional part, display only a date. If there is no integer part, display time only. Date and time display is determined by your system settings.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the long time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the short time format specified in your computer's regional settings.

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbNormal	0	Normal (default for **Dir** and **SetAttr**)
vbReadOnly	1	Read-only
vbHidden	2	Hidden
vbSystem	4	System file
vbVolume	8	Volume label
vbDirectory	16	Directory or folder
vbArchive	32	File has changed since last backup
vbAlias	64	On the Macintosh, identifier is an alias.

Only **VbNormal**, **vbReadOnly**, **vbHidden**, and **vbAlias** are available on the Macintosh.

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
Unknown	0	Drive type can't be determined.
Removable	1	Drive has removable media. This includes all floppy drives and many other varieties of storage devices.
Fixed	2	Drive has fixed (nonremovable) media. This includes all hard drives, including hard drives that are removable.
Remote	3	Network drives. This includes drives shared anywhere on a network.
CDROM	4	Drive is a CD-ROM. No distinction is made between read-only and read/write CD-ROM drives.
RAMDisk	5	Drive is a block of Random Access Memory (RAM) on the local computer that behaves like a disk drive.

File Attribute Constants

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
Normal	0	Normal file. No attributes are set.
ReadOnly	1	Read-only file. Attribute is read/write.
Hidden	2	Hidden file. Attribute is read/write.
System	4	System file. Attribute is read/write.
Volume	8	Disk drive volume label. Attribute is read-only.
Directory	16	Folder or directory. Attribute is read-only.
Archive	32	File has changed since last backup. Attribute is read/write.
Alias	64	Link or shortcut. Attribute is read-only.
Compressed	128	Compressed file. Attribute is read-only.

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
ForReading	1	Open a file for reading only. You can't write to this file.
ForWriting	2	Open a file for writing. If a file with the same name exists, its previous contents are overwritten.
ForAppending	8	Open a file and write to the end of the file.

IMEStatus Constants

The following constants can be used anywhere in your code in place of the actual values.

The constants for the Japanese locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME (default)
vbIMEModeOn	1	IME on
vbIMEModeOff	2	IME off
vbIMEModeDisable	3	IME disabled
vbIMEModeHiragana	4	Full-width Hiragana mode
vbIMEModeKatakana	5	Full-width Katakana mode
vbIMEModeKatakanaHalf	6	Half-width Katakana mode
vbIMEModeAlphaFull	7	Full-width Alphanumeric mode
vbIMEModeAlpha	8	Half-width Alphanumeric

mode

The constant values for the Korean locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME(default)
vbIMEModeAlphaFull	7	Full-width Alphanumeric mode
vbIMEModeAlpha	8	Half-width Alphanumeric mode
vbIMEModeHangulFull	9	Full-width Hangul mode
vbIMEModeHangul	10	Half-width Hangul mode

The constant values for the Chinese locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME (default)
vbIMEModeOn	1	IME on
vbIMEModeOff	2	IME off

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbKeyLButton	0x1	Left mouse button
vbKeyRButton	0x2	Right mouse button
vbKeyCancel	0x3	CANCEL key
vbKeyMButton	0x4	Middle mouse button
vbKeyBack	0x8	BACKSPACE key
vbKeyTab	0x9	TAB key
vbKeyClear	0xC	CLEAR key
vbKeyReturn	0xD	ENTER key
vbKeyShift	0x10	SHIFT key
vbKeyControl	0x11	CTRL key
vbKeyMenu	0x12	MENU key
vbKeyPause	0x13	PAUSE key
vbKeyCapital	0x14	CAPS LOCK key
vbKeyEscape	0x1B	ESC key
vbKeySpace	0x20	SPACEBAR key

vbKeyPageUp	0x21	PAGE UP key
vbKeyPageDown	0x22	PAGE DOWN key
vbKeyEnd	0x23	END key
vbKeyHome	0x24	HOME key
vbKeyLeft	0x25	LEFT ARROW key
vbKeyUp	0x26	UP ARROW key
vbKeyRight	0x27	RIGHT ARROW key
vbKeyDown	0x28	DOWN ARROW key
vbKeySelect	0x29	SELECT key
vbKeyPrint	0x2A	PRINT SCREEN key
vbKeyExecute	0x2B	EXECUTE key
vbKeySnapshot	0x2C	SNAPSHOT key
vbKeyInsert	0x2D	INSERT key
vbKeyDelete	0x2E	DELETE key
vbKeyHelp	0x2F	HELP key
vbKeyNumlock	0x90	NUM LOCK key

The A key through the Z key are the same as the ASCII equivalents A – Z:

Constant	Value	Description
vbKeyA	65	A key
vbKeyB	66	B key
vbKeyC	67	C key
vbKeyD	68	D key
vbKeyE	69	E key
vbKeyF	70	F key
vbKeyG	71	G key
vbKeyH	72	H key
vbKeyI	73	I key
vbKeyJ	74	J key
vbKeyK	75	K key
vbKeyL	76	L key

vbKeyM	77	M key
vbKeyN	78	N key
vbKeyO	79	O key
vbKeyP	80	P key
vbKeyQ	81	Q key
vbKeyR	82	R key
vbKeyS	83	S key
vbKeyT	84	T key
vbKeyU	85	U key
vbKeyV	86	V key
vbKeyW	87	W key
vbKeyX	88	X key
vbKeyY	89	Y key
vbKeyZ	90	Z key

The 0 key through 9 key are the same as their ASCII equivalents 0 – 9:

Constant	Value	Description
vbKey0	48	0 key
vbKey1	49	1 key
vbKey2	50	2 key
vbKey3	51	3 key
vbKey4	52	4 key
vbKey5	53	5 key
vbKey6	54	6 key
vbKey7	55	7 key
vbKey8	56	8 key
vbKey9	57	9 key

The following constants represent keys on the numeric keypad:

Constant	Value	Description
vbKeyNumpad0	0x60	0 key
vbKeyNumpad1	0x61	1 key
vbKeyNumpad2	0x62	2 key
vbKeyNumpad3	0x63	3 key
vbKeyNumpad4	0x64	4 key
vbKeyNumpad5	0x65	5 key
vbKeyNumpad6	0x66	6 key
vbKeyNumpad7	0x67	7 key
vbKeyNumpad8	0x68	8 key
vbKeyNumpad9	0x69	9 key
vbKeyMultiply	0x6A	MULTIPLICATION SIGN (*****) key
vbKeyAdd	0x6B	PLUS SIGN (+) key
vbKeySeparator	0x6C	ENTER key
vbKeySubtract	0x6D	MINUS SIGN (–) key
vbKeyDecimal	0x6E	DECIMAL POINT (**.**) key
vbKeyDivide	0x6F	DIVISION SIGN (/) key

The following constants represent function keys:

Constant	Value	Description
vbKeyF1	0x70	F1 key
vbKeyF2	0x71	F2 key
vbKeyF3	0x72	F3 key
vbKeyF4	0x73	F4 key
vbKeyF5	0x74	F5 key
vbKeyF6	0x75	F6 key
vbKeyF7	0x76	F7 key
vbKeyF8	0x77	F8 key
vbKeyF9	0x78	F9 key

vbKeyF10	0x79	F10 key
vbKeyF11	0x7A	F11 key
vbKeyF12	0x7B	F12 key
vbKeyF13	0x7C	F13 key
vbKeyF14	0x7D	F14 key
vbKeyF15	0x7E	F15 key
vbKeyF16	0x7F	F16 key

Miscellaneous Constants

The following constants are defined in the Visual Basic for Applications type library and can be used anywhere in your code in place of the actual values:

Constant	Equivalent	Description
vbCrLf	**Chr(**13**) + Chr(**10**)**	Carriage return–linefeed combination
vbCr	**Chr(**13**)**	Carriage return character
vbLf	**Chr(**10**)**	Linefeed character
vbNewLine	**Chr(**13**) + Chr(**10**)** or, on the Macintosh, **Chr(**13**)**	Platform-specific new line character; whichever is appropriate for current platform
vbNullChar	**Chr(**0**)**	Character having value 0
vbNullString	String having value 0	Not the same as a zero-length string (""); used for calling external procedures
vbObjectError	-2147221504	User-defined error numbers should be greater than this value. For example: `Err.Raise Number = vbObjectError + 1000`

vbTab	**Chr(**9**)**	Tab character
vbBack	**Chr(**8**)**	Backspace character
vbFormFeed	**Chr(**12**)**	Not useful in Microsoft Windows or on the Macintosh
vbVerticalTab	**Chr(**11**)**	Not useful in Microsoft Windows or on the Macintosh

The following constants can be used anywhere in your code in place of the actual values:

MsgBox Arguments

Constant	Value	Description
vbOKOnly	0	**OK** button only (default)
vbOKCancel	1	**OK** and **Cancel** buttons
vbAbortRetryIgnore	2	**Abort**, **Retry**, and **Ignore** buttons
vbYesNoCancel	3	**Yes**, **No**, and **Cancel** buttons
vbYesNo	4	**Yes** and **No** buttons
vbRetryCancel	5	**Retry** and **Cancel** buttons
vbCritical	16	Critical message
vbQuestion	32	Warning query
vbExclamation	48	Warning message
vbInformation	64	Information message
vbDefaultButton1	0	First button is default (default)
vbDefaultButton2	256	Second button is default
vbDefaultButton3	512	Third button is default
vbDefaultButton4	768	Fourth button is default

vbApplicationModal	0	Application modal message box (default)
vbSystemModal	4096	System modal message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

MsgBox Return Values

Constant	Value	Description
vbOK	1	**OK** button pressed
vbCancel	2	**Cancel** button pressed
vbAbort	3	**Abort** button pressed
vbRetry	4	**Retry** button pressed
vbIgnore	5	**Ignore** button pressed
vbYes	6	**Yes** button pressed
vbNo	7	**No** button pressed

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbFormControlMenu	0	The user chose the **Close** command from the **Control** menu on the form.
vbFormCode	1	The **Unload** statement is invoked from code.
vbAppWindows	2	The current Microsoft Windows operating environment session is ending.
vbAppTaskManager	3	The Windows **Task Manager** is closing the application.

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbHide	0	Window is hidden and focus is passed to the hidden window.
vbNormalFocus	1	Window has focus and is restored to its original size and position.
vbMinimizedFocus	2	Window is displayed as an icon with focus.
vbMaximizedFocus	3	Window is maximized with focus.
vbNormalNoFocus	4	Window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus	6	Window is displayed as an icon. The currently active window remains active.

On the Macintosh, **vbNormalFocus**, **vbMinimizedFocus**, and **vbMaximizedFocus** all place the application in the foreground; **vbHide**, **vbNoFocus**, **vbMinimizedFocus** all place the application in the background.

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
WindowsFolder	0	The Windows folder contains files installed by the Windows operating system.
SystemFolder	1	The System folder contains libraries, fonts, and device drivers.
TemporaryFolder	2	The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.

StrConv Constants

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbUpperCase	1	Converts the string to uppercase characters.
vbLowerCase	2	Converts the string to lowercase characters.
vbProperCase	3	Converts the first letter of every word in string to uppercase.
vbWide	4	Converts narrow (single-byte) characters in string to wide (double-byte) characters. Applies to Far East locales.
vbNarrow	8	Converts wide (double-byte) characters in string to narrow (single-byte) characters. Applies to Far East locales.

vbKatakana	16	Converts Hiragana characters in string to Katakana characters. Applies to Japan only.
vbHiragana	32	Converts Katakana characters in string to Hiragana characters. Applies to Japan only.
vbUnicode	64	Converts the string to Unicode using the default code page of the system. (Not available on the Macintosh.)
vbFromUnicode	128	Converts the string from Unicode to the default code page of the system. (Not available on the Macintosh.)

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbScrollBars	0x80000000	Scroll bar color
vbDesktop	0x80000001	Desktop color
vbActiveTitleBar	0x80000002	Color of the title bar for the active window
vbInactiveTitleBar	0x80000003	Color of the title bar for the inactive window
vbMenuBar	0x80000004	Menu background color
vbWindowBackground	0x80000005	Window background color
vbWindowFrame	0x80000006	Window frame color
vbMenuText	0x80000007	Color of text on menus
vbWindowText	0x80000008	Color of text in windows
vbTitleBarText	0x80000009	Color of text in caption, size box, and scroll arrow
vbActiveBorder	0x8000000A	Border color of active window
vbInactiveBorder	0x8000000B	Border color of inactive

		window
vbApplicationWorkspace	0x8000000C	Background color of multiple-document interface (MDI) applications
vbHighlight	0x8000000D	Background color of items selected in a control
vbHighlightText	0x8000000E	Text color of items selected in a control
vbButtonFace	0x8000000F	Color of shading on the face of command buttons
vbButtonShadow	0x80000010	Color of shading on the edge of command buttons
vbGrayText	0x80000011	Grayed (disabled) text
vbButtonText	0x80000012	Text color on push buttons
vbInactiveCaptionText	0x80000013	Color of text in an inactive caption
vb3DHighlight	0x80000014	Highlight color for 3-D display elements
vb3DDKShadow	0x80000015	Darkest shadow color for 3-D display elements
vb3DLight	0x80000016	Second lightest 3-D color after **vb3DHighlight**
vbInfoText	0x80000017	Color of text in ToolTips
vbInfoBackground	0x80000018	Background color of ToolTips

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions.

Constant	Value	Description
vbTrue	–1	True
vbFalse	0	False
vbUseDefault	–2	Use default setting

VarType Constants

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Description
vbEmpty	0	Uninitialized (default)
vbNull	1	Contains no valid data
vbInteger	2	Integer
vbLong	3	Long integer
vbSingle	4	Single-precision floating-point number
vbDouble	5	Double-precision floating-point number
vbCurrency	6	Currency
vbDate	7	Date
vbString	8	String
vbObject	9	Object
vbError	10	Error
vbBoolean	11	Boolean
vbVariant	12	Variant (used only for arrays of variants)
vbDataObject	13	Data access object
vbDecimal	14	Decimal
vbByte	17	Byte
vbUserDefinedType	36	Variants that contain user-defined types
vbArray	8192	Array

Visual Basic for Applications defines constants to simplify your programming. The following constants can be used anywhere in your code in place of the actual values:

Calendar Constants

CallType Constants

Color Constants

Comparison Constants

Compiler Constants

Date Constants

Date Format Constants

Dir, GetAttr, and SetAttr Constants

DriveType Constants

File Attribute Constants

Boolean Data Type

Boolean variables are stored as 16-bit (2-byte) numbers, but they can only be **True** or **False**. **Boolean** variables display as either `True` or `False` (when **Print** is used) or `#TRUE#` or `#FALSE#` (when **Write #** is used). Use the keywords **True** and **False** to assign one of the two states to **Boolean** variables.

When other numeric types are converted to **Boolean** values, 0 becomes **False** and all other values become **True**. When **Boolean** values are converted to other data types, **False** becomes 0 and **True** becomes -1.

Byte Data Type

Byte variables are stored as single, unsigned, 8-bit (1-byte) numbers ranging in value from 0–255.

The **Byte** data type is useful for containing binary data.

Currency Data Type

Currency variables are stored as 64-bit (8-byte) numbers in an integer format, 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 **Currency** is the at sign (**@**).

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

Date Data Type

Date variables are stored as IEEE 64-bit (8-byte) floating-point numbers that represent dates ranging from 1 January 100 to 31 December 9999 and times from 0:00:00 to 23:59:59. Any recognizable literal date values can be assigned to **Date** variables. Date literals must be enclosed within number signs (#), for example, `#January 1, 1993#` or `#1 Jan 93#`.

Date variables display dates according to the short date format recognized by your computer. Times display according to the time format (either 12-hour or 24-hour) recognized by your computer.

When other numeric types are converted to **Date**, values to the left of the decimal represent date information while values to the right of the decimal represent time. Midnight is 0 and midday is 0.5. Negative whole numbers represent dates before 30 December 1899.

Decimal Data Type

Decimal variables are stored as 96-bit (12-byte) signed integers scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28. With a scale of 0 (no decimal places), the largest possible value is +/-79,228,162,514,264,337,593,543,950,335. With a 28 decimal places, the largest value is +/-7.9228162514264337593543950335 and the smallest, non-zero value is +/-0.0000000000000000000000000001.

Note At this time the **Decimal** data type can only be used within a Variant, that is, you cannot declare a variable to be of type **Decimal**. You can, however, create a **Variant** whose subtype is **Decimal** using the **CDec** function.

Double Data Type

Double (double-precision floating-point) variables are stored as IEEE 64-bit (8-byte) floating-point numbers ranging in value from -1.79769313486231E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values. The type-declaration character for **Double** is the number sign (#).

Integer Data Type

Integer variables are stored as 16-bit (2-byte) numbers ranging in value from -32,768 to 32,767. The type-declaration character for **Integer** is the percent sign (**%**).

You can also use **Integer** variables 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, for example, black = 0, white = 1, and so on. It is good programming practice to

define constants using the **Const** statement for each enumerated value.

Long Data Type

Long (long integer) variables are stored as signed 32-bit (4-byte) numbers ranging in value from -2,147,483,648 to 2,147,483,647. The type-declaration character for **Long** is the ampersand (**&**).

Object Data Type

Object variables are stored as 32-bit (4-byte) addresses that refer to objects. Using the **Set** statement, a variable declared as an **Object** can have any object reference assigned to it.

Note Although a variable declared with **Object** type is flexible enough to contain a reference to any object, binding to the object referenced by that variable is always late (run-time binding). To force early binding (compile-time binding), assign the object reference to a variable declared with a specific class name.

Single Data Type

Single (single-precision floating-point) variables are stored as IEEE 32-bit (4-byte) floating-point numbers, ranging in value from -3.402823E38 to -1.401298E-45 for negative values and from 1.401298E-45 to 3.402823E38 for positive values. The type-declaration character for **Single** is the exclamation point (**!**).

String Data Type

There are two kinds of strings: variable-length and fixed-length strings.

A variable-length string can contain up to approximately 2 billion (2^31)

characters.

A fixed-length string can contain 1 to approximately 64K (2^16) characters.

Note A Public fixed-length string can't be used in a class module.

The codes for String characters range from 0–255. The first 128 characters (0–127) of the character set 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. The type-declaration character for **String** is the dollar sign (**$**).

User-Defined Data Type

Any data type you define using the **Type** statement. User-defined data types can contain one or more elements of a data type, an array, or a previously defined user-defined type. For example:

```
Type MyType
    MyName As String     ' String variable store
    MyBirthDate As Date     ' Date variable stor
    MySex As Integer     ' Integer variable stor
End Type                 ' female, 1 for male).
```

Variant Data Type

The **Variant** data type is the data type for all variables that are not explicitly declared as some other type (using statements such as **Dim**, **Private**, **Public**, or **Static**). The **Variant** data type has no type-declaration character.

A **Variant** is a special data type that can contain any kind of data except fixed-length String data. (**Variant** types now support user-defined types.) A **Variant** can also contain the special values Empty, **Error**, **Nothing**, and Null. You can determine how the data in a **Variant** is treated using the **VarType** function or **TypeName** function.

Numeric data can be any integer or real number value ranging 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 data type within the **Variant**. For example, if you assign an Integer to a **Variant**, subsequent operations treat the **Variant** as an **Integer**. However, if an arithmetic operation is performed on a **Variant** containing a Byte, 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. A **Byte** is promoted to an **Integer**, an **Integer** is promoted to a **Long**, and a **Long** and a **Single** are promoted to a Double. An error occurs when **Variant** variables containing Currency, Decimal, and **Double** values exceed their respective ranges.

You can use the **Variant** data type in place of any 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 preceding example, `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 type, the result is an arithmetic sum.

The value Empty denotes a **Variant** variable that hasn't been initialized (assigned an initial value). 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. **Null** indicates that the **Variant** variable intentionally contains no valid data.

In a **Variant**, **Error** is a special value used to indicate that an error condition has occurred in a procedure. However, unlike for other kinds of errors, normal application-level error handling does not occur. This allows you, or the application itself, to take some alternative action based on the error value. **Error** values are created by converting real numbers to error values using the **CVErr** function.

#Const Directive

Used to define conditional compiler constants for Visual Basic.

Syntax

#Const *constname* = *expression*

The **#Const** compiler directive syntax has these parts:

Part	Description
constname	Required; **Variant** (**String**). Name of the constant; follows standard variable naming conventions.
expression	Required. Literal, other conditional compiler constant, or any combination that includes any or all arithmetic or logical operators except **Is**.

Remarks

Conditional compiler constants are always Private to the module in which they appear. It is not possible to create Public compiler constants using the **#Const** directive. **Public** compiler constants can only be created in the user interface.

Only conditional compiler constants and literals can be used in *expression*. Using a standard constant defined with **Const**, or using a constant that is undefined, causes an error to occur. Conversely, constants defined using the **#Const** keyword can only be used for conditional compilation.

Conditional compiler constants are always evaluated at the module level, regardless of their placement in code.

#If...Then...#Else Directive

Conditionally compiles selected blocks of Visual Basic code.

Syntax

#If *expression* **Then**

statements

[**#ElseIf** *expression-n* **Then**

[*elseifstatements*]]

[**#Else**

[*elsestatements*]]

#End If

The **#If...Then...#Else** directive syntax has these parts:

Part	Description
expression	Required. Any expression, consisting exclusively of one or more conditional compiler constants, literals, and operators, that evaluates to **True** or **False**.
statements	Required. Visual Basic program lines or compiler directives that are evaluated if the associated expression is **True**.
expression-n	Optional. Any expression, consisting exclusively of one or more conditional compiler constants, literals, and operators, that evaluates to **True** or **False**.
elseifstatements	Optional. One or more program lines or compiler directives that are evaluated if *expression-n* is **True**.
elsestatements	Optional. One or more program lines or compiler directives that are evaluated if no previous *expression* or *expression-n* is **True**.

Remarks

The behavior of the **#If...Then...#Else** directive is the same as the **If...Then...Else** statement, except that there is no single-line form of the **#If**, **#Else**, **#ElseIf**, and **#End If** directives; that is, no other code can appear on the same line as any of the directives. Conditional compilation is typically used to compile the same program for different platforms. It is also used to prevent debugging code from appearing in an executable file. Code excluded during conditional compilation is completely omitted from the final executable file, so it has no size or performance effect.

Regardless of the outcome of any evaluation, all expressions are evaluated.

Therefore, all constants used in expressions must be defined — any undefined constant evaluates as Empty.

Note The **Option Compare** statement does not affect expressions in **#If** and **#ElseIf** statements. Expressions in a conditional-compiler directive are always evaluated with **Option Compare Text**.

Activate, Deactivate Events

The Activate event occurs when an object becomes the active window. The Deactivate event occurs when an object is no longer the active window.

Syntax

Private Sub *object*_**Activate()**

Private Sub *object*_**Deactivate()**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

An object can become active by using the **Show** method in code.

The Activate event can occur only when an object is visible. A **UserForm** loaded with **Load** isn't visible unless you use the **Show** method.

The Activate and Deactivate events occur only when you move the focus within an application. Moving the focus to or from an object in another application doesn't trigger either event.

The Deactivate event doesn't occur when unloading an object.

Initialize Event

Occurs after an object is loaded, but before it's shown.

Syntax

Private Sub *object*_**Initialize()**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The Initialize event is typically used to prepare an application or **UserForm** for use. Variables are assigned initial values, and controls may be moved or resized to accommodate initialization data.

Resize Event

Occurs when a user form is resized.

Syntax

Private Sub UserForm_Resize()

Remarks

Use a Resize event procedure to move or resize controls when the parent

UserForm is resized. You can also use this event procedure to recalculate variables or properties.

Terminate Event

Occurs when all references to an instance of an object are removed from memory by setting all variables that refer to the object to **Nothing** or when the last reference to the object goes out of scope.

Syntax

Private Sub *object*_**Terminate( )**

The *object* placeholder represents an object expression that evaluates to an

object in the Applies To list.

Remarks

The Terminate event occurs after the object is unloaded. The Terminate event isn't triggered if the instances of the **UserForm** or class are removed from memory because the application terminated abnormally. For example, if your application invokes the **End** statement before removing all existing instances of the class or **UserForm** from memory, the Terminate event isn't triggered for that class or **UserForm**.

Abs Function

Returns a value of the same type that is passed to it specifying the absolute value of a number.

Syntax

Abs(*number***)**

The required *number* argument can be any valid numeric expression. If *number* contains Null, **Null** is returned; if it is an uninitialized variable, zero is returned.

Remarks

The absolute value of a number is its unsigned magnitude. For example, `ABS(-1)` and `ABS(1)` both return `1`.

Array Function

Returns a Variant containing an array.

Syntax

Array(*arglist***)**

The required *arglist* argument is a comma-delimited list of values that are assigned to the elements of the array contained within the **Variant**. If no arguments are specified, an array of zero length is created.

Remarks

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element. In the following example, the first statement creates a variable named A as a **Variant**. The second statement assigns an array to variable A. The last statement assigns the value contained in the second array element to another variable.

```
Dim A As Variant
A = Array(10,20,30)
B = A(2)
```

The lower bound of an array created using the **Array** function is determined by the lower bound specified with the **Option Base** statement, unless **Array** is qualified with the name of the type library (for example **VBA.Array**). If qualified with the type-library name, **Array** is unaffected by **Option Base**.

Note A **Variant** that is not declared as an array can still contain an array. A **Variant** variable can contain an array of any type, except fixed-length strings and user-defined types. Although a **Variant** containing an array is conceptually different from an array whose elements are of type **Variant**, the array elements are accessed in the same way.

Asc Function

Returns an Integer representing the character code corresponding to the first letter in a string.

Syntax

Asc(*string***)**

The required *string* argument is any valid string expression. If the *string* contains no characters, a run-time error occurs.

Remarks

The range for returns is 0 – 255 on non-DBCS systems, but –32768 – 32767 on DBCS systems.

Note The **AscB** function is used with byte data contained in a string. Instead of returning the character code for the first character, **AscB** returns the first byte. The **AscW** function returns the Unicode character code except on platforms where Unicode is not supported, in which case, the behavior is identical to the **Asc** function.

Note Visual Basic for the Macintosh does not support Unicode strings. Therefore, **AscW**(*n*) cannot return all Unicode characters for n values in the range of 128 – 65,535, as it does in the Windows environment. Instead, **AscW**(*n*) attempts a "best guess" for Unicode values n greater than 127. Therefore, you should not use **AscW** in the Macintosh environment.

Atn Function

Returns a **Double** specifying the arctangent of a number.

Syntax

Atn(*number***)**

The required *number* argument is a Double or any valid numeric expression.

Remarks

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

The range of the result is -pi/2 to pi/2 radians.

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).

Choose Function

Selects and returns a value from a list of arguments.

Syntax

Choose(*index*, *choice-1*[, *choice-2*, ... [, *choice-n*]]**)**

The **Choose** function syntax has these parts:

Part	Description
index	Required. Numeric expression or field that results in a value between 1 and the number of available choices.
choice	Required. Variant expression containing one of the possible choices.

Remarks

Choose returns a value from the list of choices based on the value of *index*. If *index* is 1, **Choose** returns the first choice in the list; if *index* is 2, it returns the second choice, and so on.

You can use **Choose** to look up a value in a list of possibilities. For example, if *index* evaluates to 3 and *choice-1* = "one", *choice-2* = "two", and *choice-3* = "three", **Choose** returns "three". This capability is particularly useful if *index* represents the value in an option group.

Choose evaluates every choice in the list, even though it returns only one. For this reason, you should watch for undesirable side effects. For example, if you use the **MsgBox** function as part of an expression in all the choices, a message box will be displayed for each choice as it is evaluated, even though **Choose** returns the value of only one of them.

The **Choose** function returns a Null if *index* is less than 1 or greater than the number of choices listed.

If *index* is not a whole number, it is rounded to the nearest whole number before being evaluated.

Chr Function

Returns a String containing the character associated with the specified character code.

Syntax

Chr(*charcode***)**

The required *charcode* argument is a Long that identifies a character.

Remarks

Numbers from 0 – 31 are the same as standard, nonprintable ASCII codes. For example, **Chr(**10**)** returns a linefeed character. The normal range for *charcode* is 0 – 255. However, on DBCS systems, the actual range for *charcode* is -32768 to 65535.

Note The **ChrB** function is used with byte data contained in a **String**. Instead of returning a character, which may be one or two bytes, **ChrB** always returns a single byte. The **ChrW** function returns a **String** containing the Unicode character except on platforms where Unicode is not supported, in which case, the behavior is identical to the **Chr** function.

Note Visual Basic for the Macintosh does not support Unicode strings. Therefore, **ChrW**(*n*) cannot return all Unicode characters for n values in the range of 128 – 65,535, as it does in the Windows environment. Instead, **ChrW**(*n*) attempts a "best guess" for Unicode values n greater than 127. Therefore, you should not use **ChrW** in the Macintosh environment.

Command Function

Returns the argument portion of the command line used to launch Microsoft Visual Basic or an executable program developed with Visual Basic. The Visual Basic **Command** function is not available in Microsoft Office applications.

Syntax

Command

Remarks

When Visual Basic 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.

```
VB /cmd cmdlineargs
```

For applications developed with Visual Basic and compiled to an .exe file, **Command** returns any arguments that appear after the name of the application on the command line. For example:

```
MyApp cmdlineargs
```

To find how command line arguments can be changed in the user interface of the application you're using, search Help for "command line arguments."

Cos Function

Returns a **Double** specifying the cosine of an angle.

Syntax

Cos(*number***)**

The required *number* argument is a Double or any valid numeric expression that expresses an angle in radians.

Remarks

The **Cos** function 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.

CreateObject Function

Creates and returns a reference to an ActiveX object.

Syntax

CreateObject(*class,[servername]***)**

The **CreateObject** function syntax has these parts:

Part	Description
class	Required; **Variant** (**String**). The application name and class of the object to create.
servername	Optional; **Variant** (**String**). The name of the network server where the object will be created. If *servername* is an empty string (""), the local machine is used.

The *class* argument uses the syntax *appname.objecttype* and has these parts:

Part	Description
appname	Required; **Variant** (**String**). The name of the application providing the object.
objecttype	Required; **Variant** (**String**). The type or class of object to create.

Remarks

Every application that supports Automation provides at least one type of object. For example, a word processing application may provide an **Application** object, a **Document** object, and a **Toolbar** object.

To create an ActiveX object, assign the object returned by **CreateObject** to an

object variable:

```
' Declare an object variable to hold the object
' reference. Dim as Object causes late binding.
Dim ExcelSheet As Object
Set ExcelSheet = CreateObject("Excel.Sheet")
```

This code starts the application creating the object, in this case, a Microsoft Excel spreadsheet. Once an object is created, you reference it in code using the object variable you defined. In the following example, you access properties and methods of the new object using the object variable, `ExcelSheet`, and other Microsoft Excel objects, including the `Application` object and the `Cells` collection.

```
' Make Excel visible through the Application ob
ExcelSheet.Application.Visible = True
' Place some text in the first cell of the shee
ExcelSheet.Application.Cells(1, 1).Value = "Thi
' Save the sheet to C:\test.xls directory.
ExcelSheet.SaveAs "C:\TEST.XLS"
' Close Excel with the Quit method on the Appli
ExcelSheet.Application.Quit
' Release the object variable.
Set ExcelSheet = Nothing
```

Declaring an object variable with the `As Object` clause creates a variable that can contain a reference to any type of object. However, access to the object through that variable is late bound; that is, the binding occurs when your program is run. To create an object variable that results in early binding, that is, binding when the program is compiled, declare the object variable with a specific class ID. For example, you can declare and create the following Microsoft Excel references:

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.WorkSheet
Set xlApp = CreateObject("Excel.Application")
```

```
Set xlBook = xlApp.Workbooks.Add
Set xlSheet = xlBook.Worksheets(1)
```

The reference through an early-bound variable can give better performance, but can only contain a reference to the class specified in the declaration.

You can pass an object returned by the **CreateObject** function to a function expecting an object as an argument. For example, the following code creates and passes a reference to a Excel.Application object:

```
Call MySub (CreateObject("Excel.Application"))
```

You can create an object on a remote networked computer by passing the name of the computer to the *servername* argument of **CreateObject**. That name is the same as the Machine Name portion of a share name: for a share named "\\MyServer\Public," *servername* is "MyServer."

Note Refer to COM documentation (see *Microsoft Developer Network*) for additional information on making an application visible on a remote networked computer. You may have to add a registry key for your application.

The following code returns the version number of an instance of Excel running on a remote computer named `MyServer`:

```
Dim xlApp As Object
Set xlApp = CreateObject("Excel.Application", "
Debug.Print xlApp.Version
```

If the remote server doesn't exist or is unavailable, a run-time error occurs.

Note Use **CreateObject** when there is no current instance of the object. If an instance of the object is already running, a new instance is started, and an object of the specified type is created. To use the current instance, or to start the application and have it load a file, use the **GetObject** function.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times **CreateObject** is executed.

Returns a **Variant** (**String**) representing the current path.

Syntax

CurDir[**(***drive***)**]

The optional *drive* argument is a string expression that specifies an existing drive. If no drive is specified or if *drive* is a zero-length string (""), **CurDir** returns the path for the current drive. On the Macintosh, **CurDir** ignores any *drive* specified and simply returns the path for the current drive.

CVErr Function

Returns a Variant of subtype **Error** containing an error number specified by the user.

Syntax

CVErr(*errornumber***)**

The required *errornumber* argument is any valid error number.

Remarks

Use the **CVErr** function to create user-defined errors in user-created procedures. For example, if you create a function that accepts several arguments and normally returns a string, you can have your function evaluate the input arguments to ensure they are within acceptable range. If they are not, it is likely your function will not return what you expect. In this event, **CVErr** allows you to return an error number that tells you what action to take.

Note that implicit conversion of an **Error** is not allowed. For example, you can't directly assign the return value of **CVErr** to a variable that is not a **Variant**. However, you can perform an explicit conversion (using **CInt**, **CDbl**, and so on) of the value returned by **CVErr** and assign that to a variable of the appropriate data type.

Returns a **Variant** (**Date**) containing the current system date.

Syntax

Date

Remarks

To set the system date, use the **Date** statement.

Date, and if the calendar is Gregorian, **Date$** behavior is unchanged by the **Calendar** property setting. If the calendar is Hijri, **Date$** returns a 10-character string of the form *mm-dd-yyyy*, where *mm* (01-12), *dd* (01-30) and *yyyy* (1400-1523) are the Hijri month, day and year. The equivalent Gregorian range is Jan 1, 1980 through Dec 31, 2099.

DateAdd Function

Returns a **Variant** (**Date**) containing a date to which a specified time interval has been added.

Syntax

DateAdd(*interval, number, date*)

The **DateAdd** function syntax has these named arguments:

Part	Description
interval	Required. String expression that is the interval of time you want to add.
number	Required. Numeric expression that is the number of intervals you want to add. It can be positive (to get dates in the future) or negative (to get dates in the past).
date	Required. **Variant** (**Date**) or literal representing date to which the interval is added.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

Remarks

You can use the **DateAdd** function to add or subtract a specified time interval

from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now.

To add days to ***date***, you can use Day of Year ("y"), Day ("d"), or Weekday ("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*** is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

If the calculated date would precede the year 100 (that is, you subtract more years than are in ***date***), an error occurs.

If ***number*** isn't a Long value, it is rounded to the nearest whole number before being evaluated.

Note The format of the return value for **DateAdd** is determined by **Control Panel** settings, not by the format that is passed in ***date*** argument.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri. If month values are names, the name must be consistent with the current **Calendar** property setting. To minimize the possibility of month names conflicting with the current **Calendar** property setting, enter numeric month values (Short Date format).

DateDiff Function

Returns a **Variant** (**Long**) specifying the number of time intervals between two specified dates.

Syntax

DateDiff(*interval, date1, date2*[*, firstdayofweek*[*, firstweekofyear*]])

The **DateDiff** function syntax has these named arguments:

Part	Description
interval	Required. String expression that is the interval of time you use to calculate the difference between ***date1*** and ***date2***.
date1, ***date2***	Required; **Variant** (**Date**). Two dates you want to use in the calculation.
firstdayofweek	Optional. A constant that specifies the first day of the week. If not specified, Sunday is assumed.
firstweekofyear	Optional. A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

The ***firstdayofweek*** argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use the NLS API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Constant	Value	Description
vbUseSystem	0	Use the NLS API setting.
vbFirstJan1	1	Start with week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the new year.
vbFirstFullWeek	3	Start with first full week of the year.

Remarks

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

To calculate the number of days between ***date1*** and ***date2***, you can use either Day of year ("y") or Day ("d"). 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 ***date2*** but not ***date1***. 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 ***date2*** if it falls on a Sunday; but it doesn't count ***date1***, 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.

The ***firstdayofweek*** argument affects calculations that use the "w" and "ww" interval symbols.

If ***date1*** or ***date2*** is a date literal, the specified year becomes a permanent part of that date. However, if ***date1*** or ***date2*** is enclosed in double quotation marks (" "), and you omit the year, the current year is inserted in your code each time the ***date1*** or ***date2*** expression is evaluated. This makes it possible to write code that can be used in different years.

When comparing December 31 to January 1 of the immediately succeeding year, **DateDiff** for Year ("yyyy") returns 1 even though only a day has elapsed.

Note For ***date1*** and ***date2***, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri.

DatePart Function

Returns a **Variant** (**Integer**) containing the specified part of a given date.

Syntax

DatePart(*interval, date*[,*firstdayofweek*[, *firstweekofyear*]])

The **DatePart** function syntax has these named arguments:

Part	Description
interval	Required. String expression that is the interval of time you want to return.
date	Required. **Variant** (**Date**) value that you want to evaluate.
firstdayofweek	Optional. A constant that specifies the first day of the week. If not specified, Sunday is assumed.
firstweekofyear	Optional. A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Settings

The ***interval*** argument has these settings:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

The ***firstdayofweek*** argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use the NLS API setting.

vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The ***firstweekofyear*** argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use the NLS API setting.
vbFirstJan1	1	Start with week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the new year.
vbFirstFullWeek	3	Start with first full week of the year.

Remarks

You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

The ***firstdayofweek*** argument affects calculations that use the "w" and "ww" interval symbols.

If *date* is a date literal, the specified year becomes a permanent part of that date. However, if *date* is enclosed in double quotation marks (" "), and you omit the year, the current year is inserted in your code each time the *date* expression is evaluated. This makes it possible to write code that can be used in different

years.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri.

The returned date part is in the time period units of the current Arabic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year value is a Hijri year.

DateSerial Function

Returns a **Variant** (**Date**) for a specified year, month, and day.

Syntax

DateSerial(*year, month, day*)

The **DateSerial** function syntax has these named arguments:

Part	Description

year	Required; **Integer**. Number between 100 and 9999, inclusive, or a numeric expression.
month	Required; **Integer**. Any numeric expression.
day	Required; **Integer**. Any numeric expression.

Remarks

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 1–31 for days and 1–12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

The following example uses numeric expressions instead of absolute date numbers. Here the **DateSerial** function returns a date that is the day before the first day (`1 - 1`), two months before August (`8 - 2`), 10 years before 1990 (`1990 - 10`); in other words, May 31, 1980.

```
DateSerial(1990 - 10, 8 - 2, 1 - 1)
```

Under Windows 98 or Windows 2000, two digit years for the ***year*** argument are interpreted based on user-defined machine settings. The default settings are that values between 0 and 29, inclusive, are interpreted as the years 2000–2029. The default values between 30 and 99 are interpreted as the years 1930–1999. For all other ***year*** arguments, use a four-digit year (for example, 1800).

Earlier versions of Windows interpret two-digit years based on the defaults described above. To be sure the function returns the proper value, use a four-digit year.

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. If any single argument is outside the range -32,768 to 32,767, an error occurs. If the date specified by the three arguments falls outside the acceptable range of dates, an error occurs.

Note For ***year***, ***month***, and ***day***, if the **Calendar** property setting is Gregorian,

the supplied value is assumed to be Gregorian. If the **Calendar** property setting is Hijri, the supplied value is assumed to be Hijri.

The returned date part is in the time period units of the current Visual Basic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year value is a Hijri year. For the argument ***year***, values between 0 and 99, inclusive, are interpreted as the years 1400-1499. For all other ***year*** values, use the complete four-digit year (for example, 1520).

DateValue Function

Returns a **Variant** (**Date**).

Syntax

DateValue(*date***)**

The required *date* argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, *date* can also be any expression that can represent a date, a time, or both a date and time, in that

range.

Remarks

If *date* is a string that includes only numbers separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the Short Date format you specified for your system. **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** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *date* is omitted, **DateValue** uses the current year from your computer's system date.

If the *date* argument includes time information, **DateValue** doesn't return it. However, if *date* includes invalid time information (such as "89:98"), an error occurs.

Note For *date*, if the **Calendar** property setting is Gregorian, the supplied date must be Gregorian. If the calendar is Hijri, the supplied date must be Hijri. If the supplied date is Hijri, the argument *date* is a **String** representing a date from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

Day Function

Returns a **Variant** (**Integer**) specifying a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax

Day(*date***)**

The required *date* argument is any Variant, numeric expression, string expression, or any combination, that can represent a date. If *date* contains Null, **Null** is returned.

Note If the **Calendar** property setting is Gregorian, the returned integer represents the Gregorian day of the month for the date argument. If the calendar is Hijri, the returned integer represents the Hijri day of the month for the date argument.

DDB Function

Returns a Double specifying the depreciation of an asset for a specific time period using the double-declining balance method or some other method you specify.

Syntax

DDB(*cost, salvage*, *life*, *period*[, *factor*])

The **DDB** function has these named arguments:

Part	Description
cost	Required. **Double** specifying initial cost of the asset.
salvage	Required. **Double** specifying value of the asset at the end of its useful life.
life	Required. **Double** specifying length of useful life of the asset.
period	Required. **Double** specifying period for which asset depreciation is calculated.
factor	Optional. Variant specifying rate at which the balance declines. If omitted, 2 (double-declining method) is assumed.

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 ***life*** and ***period*** arguments 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 to calculate depreciation for a given period:

Depreciation / ***period*** = ((***cost*** – ***salvage***) * ***factor***) / ***life***

Dir Function

Returns a **String** representing the name of a file, directory, or folder that matches a specified pattern or file attribute, or the volume label of a drive.

Syntax

Dir[**(***pathname*[**,** *attributes*]**)**]

The **Dir** function syntax has these parts:

Part	Description
pathname	Optional. String expression that specifies a file name — may include directory or folder, and drive. A zero-length string ("") is returned if *pathname* is not found.
attributes	Optional. Constant or numeric expression, whose sum specifies file attributes. If omitted, returns files that match *pathname* but have no attributes.

Settings

The *attributes* argument settings are:

Constant	Value	Description
vbNormal	0	(Default) Specifies files with no attributes.
vbReadOnly	1	Specifies read-only files in addition to files with no attributes.
vbHidden	2	Specifies hidden files in addition to files with no attributes.
VbSystem	4	Specifies system files in addition to files with no attributes. Not available on the Macintosh.
vbVolume	8	Specifies volume label; if any other attributed is specified, **vbVolume** is ignored. Not available on the Macintosh.
vbDirectory	16	Specifies directories or folders in addition to files with no attributes.
vbAlias	64	Specified file name is an alias. Available only on the Macintosh.

Note These constants are specified by Visual Basic for Applications and can be used anywhere in your code in place of the actual values.

Remarks

In Microsoft Windows, **Dir** supports the use of multiple character (*****) and single character (**?**) wildcards to specify multiple files. On the Macintosh, these characters are treated as valid file name characters and can't be used as wildcards to specify multiple files.

Since the Macintosh doesn't support the wildcards, use the file type to identify groups of files. You can use the **MacID** function to specify file type instead of using the file names. For example, the following statement returns the name of the first TEXT file in the current folder:

```
Dir("SomePath", MacID("TEXT"))
```

To iterate over all files in a folder, specify an empty string:

```
Dir("")
```

If you use the **MacID** function with **Dir** in Microsoft Windows, an error occurs.

Any *attribute* value greater than 256 is considered a **MacID** value.

You must specify *pathname* the first time you call the **Dir** function, or an error occurs. If you also specify file attributes, *pathname* must be included.

Dir returns the first file name that matches *pathname*. To get any additional file names that match *pathname*, call **Dir** again with no arguments. When no more file names match, **Dir** returns a zero-length string (""). Once a zero-length string is returned, you must specify *pathname* in subsequent calls or an error occurs. You can change to a new *pathname* without retrieving all of the file names that match the current *pathname*. However, you can't call the **Dir** function recursively. Calling **Dir** with the **vbDirectory** attribute does not continually return subdirectories.

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.

DoEvents Function

Yields execution so that the operating system can process other events.

Syntax

DoEvents( )

Remarks

The **DoEvents** function returns an Integer representing the number of open forms in stand-alone versions of Visual Basic, such as Visual Basic, Professional Edition. **DoEvents** returns zero in all other applications.

DoEvents passes control to the operating system. Control is returned after the operating system has finished processing the events in its queue and all keys in the **SendKeys** queue have been sent.

DoEvents is most useful for simple things like allowing a user to cancel a process after it has started, for example a search for a file. For long-running processes, yielding the processor is better accomplished by using a Timer or delegating the task to an ActiveX EXE component.. In the latter case, the task can continue completely independent of your application, and the operating system takes case of multitasking and time slicing.

Caution Any time you temporarily yield the processor within an event procedure, make sure the procedure is not executed again from a different part of your code before the first 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.

Environ Function

Returns the **String** associated with an operating system environment variable. Not available on the Macintosh

Syntax

Environ({***envstring*** | ***number***}**)**

The **Environ** function syntax has these named arguments:

Part	Description
envstring	Optional. String expression containing the name of an environment variable.
number	Optional. Numeric expression corresponding to the numeric order of the environment string in the environment-string table. The ***number*** argument can be any numeric expression, but is rounded to a whole number before it is evaluated.

Remarks

If ***envstring*** can't be found in the environment-string table, a zero-length string ("") is returned. Otherwise, **Environ** returns the text assigned to the specified ***envstring***; that is, the text following the equal sign (=) in the environment-string table for that environment variable.

If you specify ***number***, the string occupying that numeric position in the environment-string table is returned. In this case, **Environ** returns all of the text, including ***envstring***. If there is no environment string in the specified position, **Environ** returns a zero-length string.

EOF Function

Returns an Integer containing the Boolean value **True** when the end of a file opened for **Random** or sequential **Input** has been reached.

Syntax

EOF(*filenumber***)**

The required *filenumber* argument is an **Integer** containing any valid file number.

Remarks

Use **EOF** to avoid the error generated by attempting to get input past the end of a file.

The **EOF** function returns **False** until the end of the file has been reached. With files opened for **Random** or **Binary** access, **EOF** returns **False** until the last executed **Get** statement is unable to read an entire record.

With files opened for **Binary** access, an attempt to read through the file using the **Input** function until **EOF** returns **True** generates an error. Use the **LOF** and **Loc** functions instead of **EOF** when reading binary files with **Input**, or use **Get** when using the **EOF** function. With files opened for **Output**, **EOF** always returns **True**.

Returns the error message that corresponds to a given error number.

Syntax

Error[**(***errornumber***)**]

The optional *errornumber* argument can be any valid error number. If *errornumber* is a valid error number, but is not defined, **Error** returns the string "Application-defined or object-defined error." If *errornumber* is not valid, an

error occurs. If *errornumber* is omitted, the message corresponding to the most recent run-time error is returned. If no run-time error has occurred, or *errornumber* is 0, **Error** returns a zero-length string ("").

Remarks

Examine the property settings of the **Err** object to identify the most recent run-time error. The return value of the **Error** function corresponds to the **Description** property of the **Err** object.

Exp Function

Returns a **Double** specifying *e* (the base of natural logarithms) raised to a power.

Syntax

Exp(*number***)**

The required *number* argument is a Double or any valid numeric expression.

Remarks

If the value of *number* exceeds 709.782712893, an error occurs. The constant *e* is approximately 2.718282.

Note The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.

FileAttr Function

Returns a Long representing the file mode for files opened using the **Open** statement.

Syntax

FileAttr(*filenumber*, *returntype*)

The **FileAttr** function syntax has these named arguments:

Part	Description
filenumber	Required; Integer. Any valid file number.
returntype	Required; **Integer**. Number indicating the type of information to return. Specify 1 to return a value indicating the file mode. On 16-bit systems only, specify 2 to retrieve an operating system file handle. ***Returntype*** 2 is not supported in 32-bit systems and causes an error.

Return Values

When the ***returntype*** argument is 1, the following return values indicate the file access mode:

Mode	Value
Input	1
Output	2
Random	4
Append	8
Binary	32

Returns a **Variant** (**Date**) that indicates the date and time when a file was created or last modified.

Syntax

FileDateTime(*pathname***)**

The required *pathname* argument is a string expression that specifies a file name. The *pathname* may include the directory or folder, and the drive.

FileLen Function

Returns a Long specifying the length of a file in bytes.

Syntax

FileLen(*pathname***)**

The required *pathname* argument is a string expression that specifies a file. The *pathname* may include the directory or folder, and the drive.

Remarks

If the specified file is open when the **FileLen** function is called, the value returned represents the size of the file immediately before it was opened.

Note To obtain the length of an open file, use the **LOF** function.

Description

Returns a zero-based array containing subset of a string array based on a specified filter criteria.

Syntax

Filter(*sourcesrray, match*[,** *include*[**,** *compare*]])**

The **Filter** function syntax has these named argument:

Part	Description
sourcearray	Required. One-dimensional array of strings to be searched.
match	Required. String to search for.
include	Optional. **Boolean** value indicating whether to return substrings that include or exclude ***match***. If ***include*** is **True**, **Filter** returns the subset of the array that contains ***match*** as a substring. If ***include*** is **False**, **Filter** returns the subset of the array that does not contain ***match*** as a substring.
compare	Optional. Numeric value indicating the kind of string

comparison to use. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
vbUseCompareOption	–1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Remarks

If no matches of ***match*** are found within ***sourcearray***, **Filter** returns an empty array. An error occurs if ***sourcearray*** is **Null** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

Returns the integer portion of a number.

Syntax

Int(*number***)**

Fix(*number***)**

The required *number* argument is a Double or any valid numeric expression. If

number contains Null, **Null** is returned.

Remarks

Both **Int** and **Fix** remove the fractional part of *number* and return the resulting integer value.

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))
```

Format Function

Returns a **Variant** (**String**) containing an expression formatted according to instructions contained in a format expression.

Syntax

Format(*expression*[, *format*[, *firstdayofweek*[, *firstweekofyear*]]])

The **Format** function syntax has these parts:

Part	Description

expression	Required. Any valid expression.
format	Optional. A valid named or user-defined format expression.
firstdayofweek	Optional. A constant that specifies the first day of the week.
firstweekofyear	Optional. A constant that specifies the first week of the year.

Settings

The *firstdayofweek* argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
VbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *firstweekofyear* argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
vbFirstJan1	1	Start with week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the year.
vbFirstFullWeek	3	Start with the first full week of the year.

Remarks

To Format	Do This
Numbers	Use predefined named numeric formats or create user-defined numeric formats.
Dates and times	Use predefined named date/time formats or create user-defined date/time formats.
Date and time serial numbers	Use date and time formats or numeric formats.
Strings	Create your own user-defined string formats.

If you try to format a number without specifying *format*, **Format** provides functionality similar to the **Str** function, although it is internationally aware. However, positive numbers formatted as strings using **Format** don't include a leading space reserved for the sign of the value; those converted using **Str** retain the leading space.

If you are formatting a non-localized numeric string, you should use a user-defined numeric format to ensure that you get the look you want.

Note If the **Calendar** property setting is Gregorian and *format* specifies date formatting, the supplied *expression* must be Gregorian. If the Visual Basic **Calendar** property setting is Hijri, the supplied *expression* must be Hijri.

If the calendar is Gregorian, the meaning of *format* expression symbols is unchanged. If the calendar is Hijri, all date format symbols (for example, *dddd*, *mmmm*, *yyyy*) have the same meaning but apply to the Hijri calendar. Format symbols remain in English; symbols that result in text display (for example, AM and PM) display the string (English or Arabic) associated with that symbol. The range of certain symbols changes when the calendar is Hijri.

Symbol	Range
d	1-30
dd	1-30
ww	1-51

mmm	Displays full month names (Hijri month names have no abbreviations).
y	1-355
yyyy	100-9666

Description

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

Syntax

FormatCurrency(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatCurrency** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is –1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section

	for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	–1	True
vbFalse	0	False
vbUseDefault	–2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

The position of the currency symbol relative to the currency value is determined by the system's regional settings.

Note All settings information comes from the **Regional Settings Currency** tab, except leading zero which comes from the **Number** tab.

Description

Returns an expression formatted as a date or time.

Syntax

FormatDateTime(*Date*[**,***NamedFormat*]**)**

The **FormatDateTime** function syntax has these parts:

Part	Description
Date	Required. Date expression to be formatted.
NamedFormat	Optional. Numeric value that indicates the date/time format used. If omitted, **vbGeneralDate** is used.

Settings

The *NamedFormat* argument has the following settings:

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. If there is a date part, display it as a short date. If there is a time

		part, display it as a long time. If present, both parts are displayed.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the 24-hour format (hh:mm).

Description

Returns an expression formatted as a number.

Syntax

FormatNumber(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatNumber** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is –1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.

UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	–1	True
vbFalse	0	False
vbUseDefault	–2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the **Regional Settings Number** tab.

Description

Returns an expression formatted as a percentage (multipled by 100) with a trailing % character.

Syntax

FormatPercent(*Expression*[**,***NumDigitsAfterDecimal* [**,***IncludeLeadingDigit* [**,***UseParensForNegativeNumbers* [**,***GroupDigits*]]]]**)**

The **FormatPercent** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is –1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section

	for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	–1	True
vbFalse	0	False
vbUseDefault	–2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note All settings information comes from the **Regional Settings Number** tab.

FreeFile Function

Returns an Integer representing the next file number available for use by the **Open** statement.

Syntax

FreeFile[**(***rangenumber***)**]

The optional *rangenumber* argument is a Variant that specifies the range from which the next free file number is to be returned. Specify a 0 (default) to return a

file number in the range 1 – 255, inclusive. Specify a 1 to return a file number in the range 256 – 511.

Remarks

Use **FreeFile** to supply a file number that is not already in use.

FV Function

Returns a Double specifying the future value of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

FV(*rate*, *nper*, *pmt*[, *pv*[, *type*]])

The **FV** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. Integer specifying 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	Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Optional. Variant specifying present value (or lump sum) of a series of future payments. 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. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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*** and ***nper*** arguments 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.

GetAllSettings Function

Returns a list of key settings and their respective values (originally created with **SaveSetting**) from an application's entry in the Windows registry or (on the

Macintosh) information in the application's initialization file.

Syntax

GetAllSettings(*appname, section*)

The **GetAllSettings** function syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project whose key settings are requested. On the Macintosh, this is the filename of the initialization file in the Preferences folder in the System folder.
section	Required. String e**xpression** containing the name of the section whose key settings are requested. **GetAllSettings** returns a Variant whose contents is a two-dimensional array of strings containing all the key settings in the specified section and their corresponding values.

Remarks

GetAllSettings returns an uninitialized **Variant** if either ***appname*** or ***section*** does not exist.

GetAttr Function

Returns an **Integer** representing the attributes of a file, directory, or folder.

Syntax

GetAttr(*pathname***)**

The required *pathname* argument is a string expression that specifies a file name. The *pathname* may include the directory or folder, and the drive.

Return Values

The value returned by **GetAttr** is the sum of the following attribute values:

Constant	Value	Description
vbNormal	0	Normal.
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file. Not available on the Macintosh.
vbDirectory	16	Directory or folder.
vbArchive	32	File has changed since last backup. Not available on the Macintosh.
vbAlias	64	Specified file name is an alias. Available only on the Macintosh.

Note These constants are specified by Visual Basic for Applications. The names can be used anywhere in your code in place of the actual values.

Remarks

To determine which attributes are set, use the **And** operator to perform a bitwise comparison of the value returned by the **GetAttr** function and the value of the individual file attribute you want. If the result is not zero, that attribute is set for the named file. For example, the return value of the following **And** expression is zero if the Archive attribute is not set:

```
Result = GetAttr(FName) And vbArchive
```

A nonzero value is returned if the Archive attribute is set.

GetObject Function

Returns a reference to an object provided by an ActiveX component.

Syntax

GetObject([*pathname*] [*, class*])

The **GetObject** function syntax has these named arguments:

Part	Description
pathname	Optional; **Variant** (**String**). The full path and name of the file containing the object to retrieve. If ***pathname*** is omitted, ***class*** is required.
class	Optional; **Variant** (**String**). A string representing the class of the object.

The ***class*** argument uses the syntax *appname***.***objecttype* and has these parts:

Part	Description
appname	Required; **Variant** (**String**). The name of the application providing the object.
objecttype	Required; **Variant** (**String**). The type or class of object to create.

Remarks

Use the **GetObject** function to access an ActiveX object from a file and assign the object to an object variable. Use the **Set** statement to assign the object returned by **GetObject** to the object variable. For example:

```
Dim CADObject As Object
Set CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

When this code is executed, the application associated with the specified ***pathname*** is started and the object in the specified file is activated.

If ***pathname*** is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the ***pathname*** argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate part of a file. Add an exclamation point

(**!**) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called `SCHEMA.CAD`:

```
Set LayerObject = GetObject("C:\CAD\SCHEMA.CAD!
```

If you don't specify the object's ***class***, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an **Application** object, a **Drawing** object, and a **Toolbar** object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional ***class*** argument. For example:

```
Dim MyObject As Object
Set MyObject = GetObject("C:\DRAWINGS\SAMPLE.DR
```

In the example, `FIGMENT` is the name of a drawing application and `DRAWING` is one of the object types it supports.

Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable `MyObject`. For example:

```
MyObject.Line 9, 90
MyObject.InsertText 9, 100, "Hello, world."
MyObject.SaveAs "C:\DRAWINGS\SAMPLE.DRW"
```

Note Use the **GetObject** function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the **CreateObject** function.

If an object has registered itself as a single-instance object, only one instance of

the object is created, no matter how many times **CreateObject** is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the ***pathname*** argument is omitted. You can't use **GetObject** to obtain a reference to a class created with Visual Basic.

GetSetting Function

Returns a key setting value from an application's entry in the Windows registry or (on the Macintosh) information in the application’s initialization file.

Syntax

GetSetting(*appname*, *section*, *key*[, *default*])

The **GetSetting** function syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project whose key setting is requested. On the Macintosh, this is the filename of the initialization file in the Preferences folder in the System folder.
section	Required. String expression containing the name of the section where the key setting is found.
key	Required. String expression containing the name of the key setting to return.
default	Optional. Expression containing the value to return if no value is set in the key setting. If omitted, ***default*** is assumed to be a zero-length string ("").

Remarks

If any of the items named in the **GetSetting** arguments do not exist, **GetSetting** returns the value of ***default***.

Hex Function

Returns a String representing the hexadecimal value of a number.

Syntax

Hex(*number*)

The required *number* argument is any valid numeric expression or string expression.

Remarks

If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *number* is	Hex returns
Null	Null
Empty	Zero (0)
Any other number	Up to eight hexadecimal characters

You can represent hexadecimal numbers directly by preceding numbers in the proper range with `&H`. For example, `&H10` represents decimal 16 in hexadecimal notation.

Hour Function

Returns a **Variant** (**Integer**) specifying a whole number between 0 and 23, inclusive, representing the hour of the day.

Syntax

Hour(*time***)**

The required *time* argument is any Variant, numeric expression, string expression, or any combination, that can represent a time. If *time* contains Null, **Null** is returned.

IIf

Function

Returns one of two parts, depending on the evaluation of an expression.

Syntax

IIf(*expr*, *truepart*, *falsepart*)

The **IIf** function syntax has these named arguments:

Part	Description
expr	Required. Expression you want to evaluate.
truepart	Required. Value or expression returned if ***expr*** is **True**.
falsepart	Required. Value or expression returned if ***expr*** is **False**.

Remarks

IIf always 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**.

IMEStatus Function

Returns an Integer specifying the current Input Method Editor (IME) mode of Microsoft Windows; available in East Asian versions only.

Syntax

IMEStatus

Return Values

The return values for the Japanese locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME (default)
vbIMEModeOn	1	IME on
vbIMEModeOff	2	IME off

vbIMEModeDisable	3	IME disabled
vbIMEModeHiragana	4	Full-width Hiragana mode
vbIMEModeKatakana	5	Full-width Katakana mode
vbIMEModeKatakanaHalf	6	Half-width Katakana mode
vbIMEModeAlphaFull	7	Full-width Alphanumeric mode
vbIMEModeAlpha	8	Half-width Alphanumeric mode

The return values for the Korean locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME(default)
vbIMEModeAlphaFull	7	Full-width Alphanumeric mode
vbIMEModeAlpha	8	Half-width Alphanumeric mode
vbIMEModeHangulFull	9	Full-width Hangul mode
vbIMEModeHangul	10	Half-width Hangul mode

The return values for the Chinese locale are as follows:

Constant	Value	Description
vbIMEModeNoControl	0	Don't control IME (default)
vbIMEModeOn	1	IME on
vbIMEModeOff	2	IME off

Input Function

Returns String containing characters from a file opened in **Input** or **Binary** mode.

Syntax

Input(*number*, [#]*filenumber*)

The **Input** function syntax has these parts:

Part	Description

number	Required. Any valid numeric expression specifying the number of characters to return.
filenumber	Required. Any valid file number.

Remarks

Data read with the **Input** function is usually written to a file with **Print #** or **Put**. Use this function only with files opened in **Input** or **Binary** mode.

Unlike the **Input #** statement, the **Input** function returns all of the characters it reads, including commas, carriage returns, linefeeds, quotation marks, and leading spaces.

With files opened for **Binary** access, an attempt to read through the file using the **Input** function until **EOF** returns **True** generates an error. Use the **LOF** and **Loc** functions instead of **EOF** when reading binary files with **Input**, or use **Get** when using the **EOF** function.

Note Use the **InputB** function for byte data contained within text files. With **InputB**, *number* specifies the number of bytes to return rather than the number of characters to return.

InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a String containing the contents of the text box.

Syntax

InputBox(*prompt*[, *title*] [, *default*] [, *xpos*] [, *ypos*] [, *helpfile*, *context*])

The **InputBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of ***prompt*** is approximately 1024 characters, depending on the width of the characters used. If ***prompt*** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**)** & **Chr(**10**)**) between each line.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit ***title***, the application name is placed in the title bar.
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit ***default***, the text box is displayed empty.
xpos	Optional. Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If ***xpos*** is omitted, the dialog box is horizontally centered.
ypos	Optional. Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If ***ypos*** is omitted, the dialog box is vertically

	positioned approximately one-third of the way down the screen.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If ***helpfile*** is provided, ***context*** must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If ***context*** is provided, ***helpfile*** must also be provided.

Remarks

When both ***helpfile*** and ***context*** are provided, the user can press F1 (Windows) or HELP (Macintosh) to view the Help topic corresponding to the ***context***. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box. If the user clicks **OK** or presses ENTER , the **InputBox** function returns whatever is in the text box. If the user clicks **Cancel**, the function returns a zero-length string ("").

Note To specify more than the first named argument, you must use **InputBox** in an expression. To omit some positional arguments, you must include the corresponding comma delimiter.

InStr Function

Returns a **Variant** (**Long**) specifying the position of the first occurrence of one string within another.

Syntax

InStr([*start*,]*string1*, *string2*[, *compare*])

The **InStr** function syntax has these arguments:

Part	Description
start	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If ***start*** contains Null, an error occurs. The ***start*** argument is required if ***compare*** is specified.
string1	Required. String expression being searched.
string2	Required. String expression sought.
compare	Optional. Specifies the type of string comparison. If ***compare*** is Null, an error occurs. If ***compare*** is omitted, the **Option Compare** setting determines the type of comparison. Specify a valid LCID (LocaleID) to use locale-specific rules in the comparison.

Settings

The *compare* argument settings are:

Constant	Value	Description
vbUseCompareOption	-1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Return Values

If	InStr returns

string1 is zero-length	0
string1 is **Null**	Null
string2 is zero-length	***start***
string2 is **Null**	Null
string2 is not found	0
string2 is found within ***string1***	Position at which match is found
start > ***string2***	0

Remarks

The **InStrB** function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, **InStrB** returns the byte position.

InStrRev Function

Description

Returns the position of an occurrence of one string within another, from the end of string.

Syntax

InstrRev(*stringcheck, stringmatch*[*, start*[*, compare*]])

The **InstrRev** function syntax has these named arguments:

Part	Description
stringcheck	Required. String expression being searched.
stringmatch	Required. String expression being searched for.
start	Optional. Numeric expression that sets the starting position for each search. If omitted, –1 is used, which means that the search begins at the last character position. If ***start*** contains Null, an error occurs.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
vbUseCompareOption	–1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Return Values

InStrRev returns the following values:

If	InStrRev returns
stringcheck is zero-length	0
stringcheck is **Null**	**Null**

stringmatch is zero-length	*start*
stringmatch is **Null**	**Null**
stringmatch is not found	0
stringmatch is found within ***stringcheck***	Position at which match is found
start > **Len(*stringmatch*)**	0

Remarks

Note that the syntax for the **InstrRev** function is not the same as the syntax for the **Instr** function.

IPmt Function

Returns a Double specifying the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

IPmt(*rate*, *per*, *nper*, *pv*[, *fv*[, *type*]])

The **IPmt** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. **Double** specifying payment period in the range 1 through ***nper***.
nper	Required. **Double** specifying 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	Required. **Double** specifying present value, or value today, of a series of future payments or receipts. 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	Optional. Variant specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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*** and ***nper*** arguments 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.

IRR Function

Returns a Double specifying the internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax

IRR(*values*()[, ***guess***]**)**

The **IRR** function has these named arguments:

Part	Description
***values*()**	Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).
guess	Optional. Variant specifying value you estimate will be returned by **IRR**. If omitted, ***guess*** is 0.1 (10 percent).

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 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 fixed, as it is 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 **IRR** can't find a result after 20 tries, it fails.

IsArray Function

Returns a **Boolean** value indicating whether a variable is an array.

Syntax

IsArray(*varname***)**

The required *varname* argument is an identifier specifying a variable.

Remarks

IsArray returns **True** if the variable is an array; otherwise, it returns **False**. **IsArray** is especially useful with variants containing arrays.

IsDate Function

Returns a **Boolean** value indicating whether an expression can be converted to a date.

Syntax

IsDate(*expression***)**

The required *expression* argument is a Variant containing a date expression or string expression recognizable as a date or time.

Remarks

IsDate returns **True** if the expression is a date or is recognizable as a valid date; otherwise, it returns **False**. In Microsoft Windows, the range of valid dates is January 1, 100 A.D. through December 31, 9999 A.D.; the ranges vary among operating systems.

IsEmpty Function

Returns a **Boolean** value indicating whether a variable has been initialized.

Syntax

IsEmpty(*expression***)**

The required *expression* argument is a Variant containing a numeric or string expression. However, because **IsEmpty** is used to determine if individual variables are initialized, the *expression* argument is most often a single variable name.

Remarks

IsEmpty returns **True** if the variable is uninitialized, or is explicitly set to Empty; otherwise, it returns **False**. **False** is always returned if *expression* contains more than one variable. **IsEmpty** only returns meaningful information for variants.

Returns a **Boolean** value indicating whether an expression is an error value.

Syntax

IsError(*expression***)**

The required *expression* argument can be any valid expression.

Remarks

Error values are created by converting real numbers to error values using the **CVErr** function. The **IsError** function is used to determine if a numeric expression represents an error. **IsError** returns **True** if the *expression* argument indicates an error; otherwise, it returns **False**.

IsMissing Function

Returns a **Boolean** value indicating whether an optional **Variant** argument has been passed to a procedure.

Syntax

IsMissing(*argname***)**

The required *argname* argument contains the name of an optional **Variant** procedure argument.

Remarks

Use the **IsMissing** function to detect whether or not optional **Variant** arguments have been provided in calling a procedure. **IsMissing** returns **True** if no value has been passed for the specified argument; otherwise, it returns **False**. If **IsMissing** returns **True** for an argument, use of the missing argument in other code may cause a user-defined error. If **IsMissing** is used on a **ParamArray** argument, it always returns **False**. To detect an empty **ParamArray**, test to see if the array's upper bound is less than its lower bound.

Note **IsMissing** does not work on simple data types (such as **Integer** or **Double**) because, unlike **Variants**, they don't have a provision for a "missing" flag bit. Because of this, the syntax for typed optional arguments allows you to specify a default value. If the argument is omitted when the procedure is called, then the argument will have this default value, as in the example below:

```
Sub MySub(Optional MyVar As String = "specialva
    If MyVar = "specialvalue" Then
        ' MyVar was omitted.
    Else
    ...
End Sub
```

In many cases you can omit the `If MyVar` test entirely by making the default value equal to the value you want `MyVar` to contain if the user omits it from the function call. This makes your code more concise and efficient.

IsNull Function

Returns a **Boolean** value that indicates whether an expression contains no valid data (Null).

Syntax

IsNull(*expression***)**

The required *expression* argument is a Variant containing a numeric expression or string expression.

Remarks

IsNull returns **True** if *expression* is **Null**; otherwise, **IsNull** returns **False**. If *expression* consists of more than one variable, **Null** in any constituent variable causes **True** to be returned for the entire expression.

The **Null** value indicates that the **Variant** contains no valid data. **Null** is not the same as Empty, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (""), which is sometimes referred to as a null string.

Important Use the **IsNull** function to determine whether an expression contains a **Null** value. Expressions that you might expect to evaluate to **True** under some circumstances, such as `If Var = Null` and `If Var <> Null`, are always **False**. This is because any expression containing a **Null** is itself **Null** and, therefore, **False**.

IsNumeric Function

Returns a **Boolean** value indicating whether an expression can be evaluated as a number.

Syntax

IsNumeric(*expression***)**

The required *expression* argument is a Variant containing a numeric expression or string expression.

Remarks

IsNumeric returns **True** if the entire *expression* is recognized as a number; otherwise, it returns **False**.

IsNumeric returns **False** if *expression* is a date expression.

IsObject Function

Returns a **Boolean** value indicating whether an identifier represents an object variable.

Syntax

IsObject(*identifier***)**

The required *identifier* argument is a variable name.

Remarks

IsObject is useful only in determining whether a Variant is of **VarType vbObject**. This could occur if the **Variant** actually references (or once referenced) an object, or if it contains **Nothing.**

IsObject returns **True** if *identifier* is a variable declared with Object type or any valid class type, or if *identifier* is a **Variant** of **VarType vbObject**, or a user-defined object; otherwise, it returns **False**. **IsObject** returns **True** even if the variable has been set to **Nothing**.

Use error trapping to be sure that an object reference is valid.

Join

Function

Description

Returns a string created by joining a number of substrings contained in an array.

Syntax

Join(*sourcearray*[, *delimiter*])

The **Join** function syntax has these named arguments:

Part	Description
sourcearray	Required. One-dimensional array containing substrings to be joined.
delimiter	Optional. String character used to separate the substrings in the returned string. If omitted, the space character (" ") is used. If ***delimiter*** is a zero-length string (""), all items in the list are

concatenated with no delimiters.

LBound Function

Returns a Long containing the smallest available subscript for the indicated dimension of an array.

Syntax

LBound(*arrayname*[**,** *dimension*]**)**

The **LBound** function syntax has these parts:

Part	Description

arrayname	Required. Name of the array variable; follows standard variable naming conventions.
dimension	Optional; **Variant** (**Long**). Whole number indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If *dimension* is omitted, 1 is assumed.

Remarks

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 returns the values in the following table for an array with the following dimensions:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

Statement	Return Value
`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. The base of an array created with the **Array** function is zero; it is unaffected by **Option Base**.

Arrays for which dimensions are set using the **To** clause in a **Dim**, **Private**, **Public**, **ReDim**, or **Static** statement can have any integer value as a lower bound.

LCase Function

Returns a String that has been converted to lowercase.

Syntax

LCase(*string*)

The required *string* argument is any valid string expression. If *string* contains Null, Null is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

Left Function

Returns a **Variant** (**String**) containing a specified number of characters from the left side of a string.

Syntax

Left(***string***, ***length***)

The **Left** function syntax has these named arguments:

Part	Description
string	Required. String expression from which the leftmost characters are returned. If ***string*** contains Null, Null is returned.
length	Required; **Variant** (**Long**). Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in ***string***, the entire string is returned.

Remarks

To determine the number of characters in ***string***, use the **Len** function.

Note Use the **LeftB** function with byte data contained in a string. Instead of specifying the number of characters to return, ***length*** specifies the number of bytes.

Len Function

Returns a Long containing the number of characters in a string or the number of bytes required to store a variable.

Syntax

Len(*string* | *varname*)

The **Len** function syntax has these parts:

Part	Description
string	Any valid string expression. If *string* contains Null, Null is returned.
Varname	Any valid variable name. If *varname* contains **Null**, **Null** is returned. If *varname* is a Variant, **Len** treats it the same as a **String** and always returns the number of characters it contains.

Remarks

One (and only one) of the two possible arguments must be specified. With user-defined types, **Len** returns the size as it will be written to the file.

Note Use the **LenB** function with byte data contained in a string, as in double-byte character set (DBCS) languages. Instead of returning the number of characters in a string, **LenB** returns the number of bytes used to represent that

string. With user-defined types, **LenB** returns the in-memory size, including any padding between elements. For sample code that uses **LenB**, see the second example in the example topic.

Note **Len** may not be able to determine the actual number of storage bytes required when used with variable-length strings in user-defined data types.

Loc Function

Returns a Long specifying the current read/write position within an open file.

Syntax

Loc(*filenumber***)**

The required *filenumber* argument is any valid Integer file number.

Remarks

The following describes the return value for each file access mode:

Mode	Return Value
Random	Number of the last record read from or written to the file.
Sequential	Current byte position in the file divided by 128. However, information returned by **Loc** for sequential files is neither used nor required.
Binary	Position of the last byte read or written.

LOF Function

Returns a Long representing the size, in bytes, of a file opened using the **Open** statement.

Syntax

LOF(*filenumber***)**

The required *filenumber* argument is an Integer containing a valid file number.

Note Use the **FileLen** function to obtain the length of a file that is not open.

Log Function

Returns a **Double** specifying the natural logarithm of a number.

Syntax

Log(*number***)**

The required *number* argument is a Double or any valid numeric expression greater than zero.

Remarks

The natural logarithm is the logarithm to the base *e*. The constant *e* is approximately 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(x)* = **Log**(*x*) / **Log**(*n*)

The following example illustrates a custom **Function** that calculates base-10 logarithms:

```
Static Function Log10(X)
    Log10 = Log(X) / Log(10#)
End Function
```

Returns a **Variant** (**String**) containing a copy of a specified string without leading spaces (**LTrim**), trailing spaces (**RTrim**), or both leading and trailing spaces (**Trim**).

Syntax

LTrim(*string*)

RTrim(*string*)

Trim(*string*)

The required *string* argument is any valid string expression. If *string* contains Null, **Null** is returned.

Mid Function

Returns a **Variant** (**String**) containing a specified number of characters from a string.

Syntax

Mid(***string***, ***start***[, ***length***])

The **Mid** function syntax has these named arguments:

Part	Description
string	Required. String expression from which characters are returned. If ***string*** contains Null, **Null** is returned.
start	Required; Long. Character position in ***string*** at which the part to be taken begins. If ***start*** is greater than the number of characters in ***string***, **Mid** returns a zero-length string ("").
length	Optional; **Variant** (**Long**). Number of characters to return. If omitted or if there are fewer than ***length*** characters in the text (including the character at ***start***), all characters from the ***start*** position to the end of the string are returned.

Remarks

To determine the number of characters in ***string***, use the **Len** function.

Note Use the **MidB** function with byte data contained in a string, as in double-byte character set languages. Instead of specifying the number of characters, the arguments specify numbers of bytes. For sample code that uses **MidB**, see the second example in the example topic.

Minute Function

Returns a **Variant** (**Integer**) specifying a whole number between 0 and 59, inclusive, representing the minute of the hour.

Syntax

Minute(*time***)**

The required *time* argument is any Variant, numeric expression, string expression, or any combination, that can represent a time. If *time* contains Null, **Null** is returned.

MIRR Function

Returns a Double specifying the modified internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax

MIRR(*values*(), *finance_rate*, *reinvest_rate*)

The **MIRR** function has these named arguments:

Part	Description
***values*()**	Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).
finance_rate	Required. **Double** specifying interest rate paid as the cost of financing.
reinvest_rate	Required. **Double** specifying interest rate received on gains from cash reinvestment.

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 (***finance_rate***) and the interest rate received on reinvestment of cash (***reinvest_rate***).

The ***finance_rate*** and ***reinvest_rate*** arguments 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.

Month Function

Returns a **Variant** (**Integer**) specifying a whole number between 1 and 12, inclusive, representing the month of the year.

Syntax

Month(*date***)**

The required *date* argument is any Variant, numeric expression, string expression, or any combination, that can represent a date. If *date* contains Null, **Null** is returned.

Note If the **Calendar** property setting is Gregorian, the returned integer represents the Gregorian day of the week for the date argument. If the calendar is Hijri, the returned integer represents the Hijri day of the week for the date argument. For Hijri dates, the argument number is any numeric expression that can represent a date and/or time from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

Description

Returns a string indicating the specified month.

Syntax

MonthName(*month*[**,** *abbreviate*]**)**

The **MonthName** function syntax has these parts:

Part	Description
month	Required. The numeric designation of the month. For example, January is 1, February is 2, and so on.
abbreviate	Optional. **Boolean** value that indicates if the month name is to be abbreviated. If omitted, the default is **False**, which means that the month name is not abbreviated.

MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and returns an **Integer** indicating which button the user clicked.

Syntax

MsgBox(*prompt*[, *buttons*] [, *title*] [, *helpfile*, *context*])

The **MsgBox** function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of ***prompt*** is approximately 1024 characters, depending on the width of the characters used. If ***prompt*** consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return – linefeed character combination (**Chr(**13**)** & **Chr(**10**)**) between each line.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for ***buttons*** is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit ***title***, the application name is placed in the title bar.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If ***helpfile*** is provided, ***context*** must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If ***context*** is provided, ***helpfile*** must also be provided.

Settings

The ***buttons*** argument settings are:

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.
vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.
vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display **Critical Message** icon.
vbQuestion	32	Display **Warning Query** icon.
vbExclamation	48	Display **Warning Message** icon.
vbInformation	64	Display **Information Message** icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the ***buttons*** argument, use only one number from each group.

Note These constants are specified by Visual Basic for Applications. As a result, the names can be used anywhere in your code in place of the actual values.

Return Values

Constant	Value	Description
vbOK	1	**OK**
vbCancel	2	**Cancel**
vbAbort	3	**Abort**
vbRetry	4	**Retry**
vbIgnore	5	**Ignore**
vbYes	6	**Yes**
vbNo	7	**No**

Remarks

When both ***helpfile*** and ***context*** are provided, the user can press F1 (Windows) or HELP (Macintosh) to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box.

If the dialog box displays a **Cancel** button, pressing the ESC key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

Note To specify more than the first named argument, you must use **MsgBox** in an expression. To omit some positional arguments, you must include the corresponding comma delimiter.

Returns a **Variant** (**Date**) specifying the current date and time according your computer's system date and time.

Syntax

Now

NPer Function

Returns a Double specifying the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

NPer(*rate*, *pmt*, *pv*[, *fv*[, *type*]])

The **NPer** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Required. **Double** specifying present value, or value today, of a series of future payments or receipts. 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	Optional. Variant specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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).

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.

NPV Function

Returns a Double specifying 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*, *values*())

The **NPV** function has these named arguments:

Part	Description

rate	Required. **Double** specifying discount rate over the length of the period, expressed as a decimal.
***values*()**	Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).

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 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 ***values*( )**.

The **NPV** function is similar to the **PV** function (present value) except that the **PV** function allows cash flows to begin either at the end or the beginning of a period. Unlike the variable **NPV** cash flow values, **PV** cash flows must be fixed throughout the investment.

Oct Function

Returns a **Variant** (**String**) representing the octal value of a number.

Syntax

Oct(*number*)

The required *number* argument is any valid numeric expression or string expression.

Remarks

If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *number* is	Oct returns
Null	**Null**
Empty	Zero (0)
Any other number	Up to 11 octal characters

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.

Partition Function

Returns a **Variant** (**String**) indicating where a number occurs within a calculated series of ranges.

Syntax

Partition(*number, start, stop, interval*)

The **Partition** function syntax has these named arguments:

Part	Description
number	Required. Whole number that you want to evaluate against the ranges.
start	Required. Whole number that is the start of the overall range of numbers. The number can't be less than 0.
stop	Required. Whole number that is the end of the overall range

of numbers. The number can't be equal to or less than ***start***.

Remarks

The **Partition** function identifies the particular range in which ***number*** falls and returns a **Variant** (**String**) describing that range. The **Partition** function is most useful in queries. You can create a select query that shows how many orders fall within various ranges, for example, order values from 1 to 1000, 1001 to 2000, and so on.

The following table shows how the ranges are determined using three sets of ***start***, ***stop***, and ***interval*** parts. The First Range and Last Range columns show what **Partition** returns. The ranges are represented by *lowervalue*:*uppervalue*, where the low end (*lowervalue*) of the range is separated from the high end (*uppervalue*) of the range 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 the table shown above, the third line 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.

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 ***stop***, 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 parts is Null, **Partition** returns a **Null**.

Pmt Function

Returns a Double specifying the payment for an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

Pmt(*rate*, *nper*, *pv*[, *fv*[, *type*]])

The **Pmt** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. Integer specifying 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	Required. **Double** specifying 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	Optional. Variant specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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*** and ***nper*** arguments 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.

PPmt Function

Returns a Double specifying the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Syntax

PPmt(*rate*, *per*, *nper*, *pv*[, *fv*[, *type*]])

The **PPmt** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. Integer specifying payment period in the range 1 through ***nper***.
nper	Required. **Integer** specifying 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	Required. **Double** specifying present value, or value today, of a series of future payments or receipts. 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	Optional. Variant specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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*** and ***nper*** arguments 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.

PV Function

Returns a Double specifying the present value of an annuity based on periodic, fixed payments to be paid in the future and a fixed interest rate.

Syntax

PV(*rate*, *nper*, *pmt*[, *fv*[, *type*]])

The **PV** function has these named arguments:

Part	Description
rate	Required. **Double** specifying 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	Required. Integer specifying 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	Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
fv	Optional. Variant specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

Remarks

An annuity is a series of fixed 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*** and ***nper*** arguments 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.

QBColor Function

Returns a Long representing the RGB color code corresponding to the specified color number.

Syntax

QBColor(*color***)**

The required *color* argument is a whole number in the range 0–15.

Settings

The *color* argument has these settings:

Number	Color	Number	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

Remarks

The *color* argument represents color values used by earlier versions of Basic (such as Microsoft Visual Basic for MS-DOS and the 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 Visual Basic for Applications.

Rate Function

Returns a Double specifying the interest rate per period for an annuity.

Syntax

Rate(*nper*, *pmt*, *pv*[, *fv*[, *type*[, *guess*]]])

The **Rate** function has these named arguments:

Part	Description

nper	Required. **Double** specifying 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	Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
pv	Required. **Double** specifying present value, or value today, of a series of future payments or receipts. 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	Optional. Variant specifying future value or cash balance you want after you make the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type	Optional. **Variant** specifying a number indicating when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.
guess	Optional. **Variant** specifying value you estimate will be returned by **Rate**. If omitted, ***guess*** is 0.1 (10 percent).

Remarks

An annuity is a series of fixed 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).

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 **Rate** can't find a result after 20 tries, it fails. If your guess is 10 percent and **Rate** fails, try a different value for ***guess***.

Replace

Function

Description

Returns a string in which a specified substring has been replaced with another substring a specified number of times.

Syntax

Replace(*expression*, *find*, *replace*[, *start*[, *count*[, *compare*]]])

The **Replace** function syntax has these named arguments:

Part	Description
expression	Required. String expression containing substring to replace.
find	Required. Substring being searched for.
replace	Required. Replacement substring.

start	Optional. Position within ***expression*** where substring search is to begin. If omitted, 1 is assumed.
count	Optional. Number of substring substitutions to perform. If omitted, the default value is –1, which means make all possible substitutions.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
vbUseCompareOption	–1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Return Values

Replace returns the following values:

If	Replace returns
expression is zero-length	Zero-length string ("")
expression is **Null**	An error.
find is zero-length	Copy of ***expression***.
replace is zero-length	Copy of ***expression*** with all occurences of *find* removed.
start > **Len(*expression*)**	Zero-length string.
count is 0	Copy of ***expression***.

Remarks

The return value of the **Replace** function is a string, with substitutions made, that begins at the position specified by ***start*** and and concludes at the end of the ***expression*** string. It is not a copy of the original string from start to finish.

RGB Function

Returns a Long whole number representing an RGB color value.

Syntax

RGB(*red, green, blue*)

The **RGB** function syntax has these named arguments:

Part	Description
red	Required; **Variant** (**Integer**). Number in the range 0–255, inclusive, that represents the red component of the color.
green	Required; **Variant** (**Integer**). Number in the range 0–255, inclusive, that represents the green component of the color.
blue	Required; **Variant** (**Integer**). Number in the range 0–255, inclusive, that represents the blue component of the color.

Remarks

Application methods and properties that accept a color specification expect that specification to be a number representing an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue to cause a specific color to be displayed.

The value for any argument to **RGB** that exceeds 255 is assumed to be 255.

The following table lists some standard colors and the red, green, and blue values they include:

Color	Red Value	Green Value	Blue Value
Black	0	0	0
Blue	0	0	255
Green	0	255	0
Cyan	0	255	255
Red	255	0	0
Magenta	255	0	255
Yellow	255	255	0
White	255	255	255

The RGB color values returned by this function are incompatible with those used

by the Macintosh operating system. They may be used within the context of Microsoft applications for the Macintosh, but should not be used when communicating color changes directly to the Macintosh operating system.

Right Function

Returns a **Variant** (**String**) containing a specified number of characters from the right side of a string.

Syntax

Right(***string***, ***length***)

The **Right** function syntax has these named arguments:

Part	Description
string	Required. String expression from which the rightmost characters are returned. If ***string*** contains Null, **Null** is returned.
length	Required; **Variant** (**Long**). Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in ***string***, the entire string is returned.

Remarks

To determine the number of characters in ***string***, use the **Len** function.

Note Use the **RightB** function with byte data contained in a string. Instead of specifying the number of characters to return, ***length*** specifies the number of bytes.

Rnd Function

Returns a **Single** containing a random number.

Syntax

Rnd[**(***number***)**]

The optional *number* argument is a Single or any valid numeric expression.

Return Values

If *number* is	Rnd generates
Less than zero	The same number every time, using *number* as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

Remarks

The **Rnd** function returns a value less than 1 but greater than or equal to zero.

The value of *number* determines how **Rnd** generates a random number:

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.

Before calling **Rnd,** use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:

```
Int((upperbound - lowerbound + 1) * Rnd + lower
```

Here, *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

Note To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.

Description

Returns a number rounded to a specified number of decimal places.

Syntax

Round(*expression* [**,***numdecimalplaces*]**)**

The **Round** function syntax has these parts:

Part	Description
expression	Required. Numeric expression being rounded.
numdecimalplaces	Optional. Number indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the **Round** function.

Second Function

Returns a **Variant** (**Integer**) specifying a whole number between 0 and 59, inclusive, representing the second of the minute.

Syntax

Second(*time***)**

The required *time* argument is any Variant, numeric expression, string expression, or any combination, that can represent a time. If *time* contains Null, **Null** is returned.

Seek Function

Returns a Long specifying the current read/write position within a file opened using the **Open** statement.

Syntax

Seek(*filenumber***)**

The required *filenumber* argument is an Integer containing a valid file number.

Remarks

Seek returns a value between 1 and 2,147,483,647 (equivalent to 2^31 – 1), inclusive.

The following describes the return values for each file access mode.

Mode	Return Value
Random	Number of the next record read or written
Binary, **Output**, **Append**, **Input**	Byte position at which the next operation takes place. The first byte in a file is at position 1, the second byte is at position 2, and so on.

Sgn

Function

Returns a **Variant** (**Integer**) indicating the sign of a number.

Syntax

Sgn(*number***)**

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

Return Values

If *number* is	Sgn returns
Greater than zero	1
Equal to zero	0
Less than zero	-1

Remarks

The sign of the *number* argument determines the return value of the **Sgn** function.

Shell Function

Runs an executable program and returns a **Variant** (**Double**) representing the program's task ID if successful, otherwise it returns zero.

Syntax

Shell(*pathname*[,*windowstyle*])

The **Shell** function syntax has these named arguments:

Part	Description

pathname	Required; **Variant** (**String**). Name of the program to execute and any required arguments or command-line switches; may include directory or folder and drive. On the Macintosh, you can use the **MacID** function to specify an application's signature instead of its name. The following example uses the signature for Microsoft Word: `Shell MacID("MSWD")`
windowstyle	Optional. **Variant** (**Integer**) corresponding to the style of the window in which the program is to be run. If ***windowstyle*** is omitted, the program is started minimized with focus. On the Macintosh (System 7.0 or later), ***windowstyle*** only determines whether or not the application gets the focus when it is run.

The ***windowstyle*** named argument has these values:

Constant	Value	Description
vbHide	0	Window is hidden and focus is passed to the hidden window. The **vbHide** constant is not applicable on Macintosh platforms.
vbNormalFocus	1	Window has focus and is restored to its original size and position.
vbMinimizedFocus	2	Window is displayed as an icon with focus.
vbMaximizedFocus	3	Window is maximized with focus.
vbNormalNoFocus	4	Window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus	6	Window is displayed as an icon. The currently active window remains active.

Remarks

If the **Shell** function successfully executes the named file, it returns the task 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.

On the Macintosh, **vbNormalFocus**, **vbMinimizedFocus**, and **vbMaximizedFocus** all place the application in the foreground; **vbHide**, **vbNoFocus**, **vbMinimizeFocus** all place the application in the background.

Note By default, the **Shell** function runs other programs asynchronously. This means that a program started with **Shell** might not finish executing before the statements following the **Shell** function are executed.

Sin Function

Returns a **Double** specifying the sine of an angle.

Syntax

Sin(*number***)**

The required *number* argument is a Double or any valid numeric expression that expresses an angle in radians.

Remarks

The **Sin** function 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.

SLN Function

Returns a Double specifying the straight-line depreciation of an asset for a single period.

Syntax

SLN(*cost*, *salvage*, *life*)

The **SLN** function has these named arguments:

Part	Description

cost	Required. **Double** specifying initial cost of the asset.
salvage	Required. **Double** specifying value of the asset at the end of its useful life.
life	Required. **Double** specifying length of the useful life of the asset.

Remarks

The depreciation period must be expressed in the same unit as the ***life*** argument. All arguments must be positive numbers.

Returns a **Variant** (**String**) consisting of the specified number of spaces.

Syntax

Space(*number*)

The required *number* argument is the number of spaces you want in the string.

Remarks

The **Space** function is useful for formatting output and clearing data in fixed-length strings.

Spc Function

Used with the **Print #** statement or the **Print** method to position output.

Syntax

Spc(*n***)**

The required *n* argument is the number of spaces to insert before displaying or printing the next expression in a list.

Remarks

If *n* is less than the output line width, the next print position immediately follows the number of spaces printed. If *n* is greater than the output line width, **Spc** calculates the next print position using the formula:

currentprintposition + (*n* **Mod** *width*)

For example, if the current print position is 24, the output line width is 80, and you specify **Spc(**90**)**, the next print will start at position 34 (current print position + the remainder of 90/80). If the difference between the current print position and the output line width is less than *n* (or *n* **Mod** *width*), the **Spc** function skips to the beginning of the next line and generates spaces equal to *n* – (*width* – *currentprintposition*).

Note Make sure your tabular columns are wide enough to accommodate wide letters.

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 than one fixed-width column.

Split Function

Description

Returns a zero-based, one-dimensional array containing a specified number of substrings.

Syntax

Split(*expression*[, *delimiter*[, *limit*[, *compare*]]])

The **Split** function syntax has these named arguments:

Part	Description
expression	Required. String expression containing substrings and delimiters. If *expression* is a zero-length string(""), **Split** returns an empty array, that is, an array with no elements and no data.
delimiter	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If ***delimiter*** is a zero-length string, a single-element array containing the entire ***expression*** string is returned.
limit	Optional. Number of substrings to be returned; –1 indicates that all substrings are returned.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
vbUseCompareOption	–1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Sqr Function

Returns a **Double** specifying the square root of a number.

Syntax

Sqr(*number***)**

The required *number* argument is a Double or any valid numeric expression greater than or equal to zero.

Str Function

Returns a **Variant** (**String**) representation of a number.

Syntax

Str(*number*)

The required *number* argument is a Long containing any valid numeric expression.

Remarks

When numbers are converted to strings, a leading space is always reserved for the sign of *number*. If *number* is positive, the returned string contains a leading space and the plus sign is implied.

Use the **Format** function to convert numeric values you want formatted as dates, times, or currency or in other user-defined formats. Unlike **Str**, the **Format** function doesn't include a leading space for the sign of *number*.

Note The **Str** function recognizes only the period (**.**) as a valid decimal separator. When different decimal separators may be used (for example, in international applications), use **CStr** to convert a number to a string.

StrComp Function

Returns a **Variant** (**Integer**) indicating the result of a string comparison.

Syntax

StrComp(***string1***, ***string2***[, ***compare***])

The **StrComp** function syntax has these named arguments:

Part	Description
string1	Required. Any valid string expression.
string2	Required. Any valid string expression.
compare	Optional. Specifies the type of string comparison. If the ***compare*** argument is Null, an error occurs. If ***compare*** is omitted, the **Option Compare** setting determines the type of comparison.

Settings

The **compare** argument settings are:

Constant	Value	Description
vbUseCompareOption	-1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Return Values

The **StrComp** function has the following return values:

If	StrComp returns
string1 is less than ***string2***	-1
string1 is equal to ***string2***	0
string1 is greater than ***string2***	1
string1 or ***string2*** is **Null**	**Null**

StrConv Function

Returns a **Variant** (**String**) converted as specified.

Syntax

StrConv(***string, conversion, LCID***)

The **StrConv** function syntax has these named arguments:

Part	Description
string	Required. String expression to be converted.
conversion	Required. Integer. The sum of values specifying the type of conversion to perform.
LCID	Optional. The LocaleID, if different than the system LocaleID. (The system LocaleID is the default.)

Settings

The ***conversion*** argument settings are:

Constant	Value	Description
vbUpperCase	1	Converts the string to uppercase characters.
vbLowerCase	2	Converts the string to lowercase characters.
vbProperCase	3	Converts the first letter of every word in string to

		uppercase.
vbWide*	4*	Converts narrow (single-byte) characters in string to wide (double-byte) characters.
vbNarrow*	8*	Converts wide (double-byte) characters in string to narrow (single-byte) characters.
vbKatakana**	16**	Converts Hiragana characters in string to Katakana characters.
vbHiragana**	32**	Converts Katakana characters in string to Hiragana characters.
vbUnicode	64	Converts the string to Unicode using the default code page of the system. (Not available on the Macintosh.)
vbFromUnicode	128	Converts the string from Unicode to the default code page of the system. (Not available on the Macintosh.)

*Applies to Far East locales.

**Applies to Japan only.

Note These constants are specified by Visual Basic for Applications. As a result, they may be used anywhere in your code in place of the actual values. Most can be combined, for example, **vbUpperCase + vbWide**, except when they are mutually exclusive, for example, **vbUnicode + vbFromUnicode**. The constants **vbWide**, **vbNarrow**, **vbKatakana**, and **vbHiragana** cause run-time errors when used in locales where they do not apply.

The following are valid word separators for proper casing: Null (**Chr$(**0**)**), horizontal tab (**Chr$(**9**)**), linefeed (**Chr$(**10**)**), vertical tab (**Chr$(**11**)**), form feed (**Chr$(**12**)**), carriage return (**Chr$(**13**)**), space (SBCS) (**Chr$(**32**)**). The actual value for a space varies by country for DBCS.

Remarks

When you're converting from a **Byte** array in ANSI format to a string, you should use the **StrConv** function. When you're converting from such an array in

Unicode format, use an assignment statement.

Description

Returns a string in which the character order of a specified string is reversed.

Syntax

StrReverse(*expression*)

The ***expression*** argument is the string whose characters are to be reversed. If ***expression*** is a zero-length string (""), a zero-length string is returned. If ***expression*** is **Null**, an error occurs.

String Function

Returns a **Variant** (**String**) containing a repeating character string of the length specified.

Syntax

String(***number***, ***character***)

The **String** function syntax has these named arguments:

Part	Description
number	Required; Long. Length of the returned string. If ***number*** contains Null, **Null** is returned.
character	Required; Variant. Character code specifying the character or string expression whose first character is used to build the return string. If ***character*** contains **Null**, **Null** is returned.

Remarks

If you specify a number for ***character*** greater than 255, **String** converts the number to a valid character code using the formula:

character **Mod** 256

Switch Function

Evaluates a list of expressions and returns a **Variant** value or an expression associated with the first expression in the list that is **True**.

Syntax

Switch(*expr-1***,** *value-1*[**,** *expr-2***,** *value-2* **…** [**,** *expr-n***,***value-n*]]**)**

The **Switch** function syntax has these parts:

Part	Description
expr	Required. Variant expression you want to evaluate.
value	Required. Value or expression to be returned if the corresponding expression is **True**.

Remarks

The **Switch** function argument list consists of pairs of expressions and values. The expressions are evaluated from left to right, and the value associated with the first expression to evaluate to **True** is returned. If the parts aren't properly paired, a run-time error occurs. For example, if *expr-1* is **True**, **Switch** returns *value-1*. If *expr-1* is **False**, but *expr-2* is **True**, **Switch** returns *value-2*, and so on.

Switch returns a Null value if:

None of the expressions is **True**.

The first **True** expression has a corresponding value 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 expression results in a division by zero error, an error occurs.

SYD Function

Returns a Double specifying the sum-of-years' digits depreciation of an asset for a specified period.

Syntax

SYD(*cost*, *salvage*, *life*, *period*)

The **SYD** function has these named arguments:

Part	Description

cost	Required. **Double** specifying initial cost of the asset.
salvage	Required. **Double** specifying value of the asset at the end of its useful life.
life	Required. **Double** specifying length of the useful life of the asset.
period	Required. **Double** specifying period for which asset depreciation is calculated.

Remarks

The ***life*** and ***period*** arguments 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.

Tab Function

Used with the **Print #** statement or the **Print** method to position output.

Syntax

Tab[**(***n***)**]

The optional *n* argument is the column number moved to before displaying or printing the next expression in a list. If omitted, **Tab** moves the insertion point to the beginning of the next print zone. This allows **Tab** to be used instead of a

comma in locales where the comma is used as a decimal separator.

Remarks

If the current print position on the current line is greater than *n*, **Tab** skips to the *n*th column on the next output line. If *n* is less than 1, **Tab** moves the print position to column 1. If *n* is greater than the output line width, **Tab** calculates the next print position using the formula:

n **Mod** *width*

For example, if *width* is 80 and you specify **Tab(**90**)**, the next print will start at column 10 (the remainder of 90/80). If *n* 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.

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.

Note Make sure your tabular columns are wide enough to accommodate wide letters.

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 than one fixed-width column.

Tan Function

Returns a **Double** specifying the tangent of an angle.

Syntax

Tan(*number***)**

The required *number* argument is a Double or any valid numeric expression that expresses an angle in radians.

Remarks

Tan 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 side adjacent to the angle.

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

Returns a **Variant** (**Date**) indicating the current system time.

Syntax

Time

Remarks

To set the system time, use the **Time** statement.

Returns a **Single** representing the number of seconds elapsed since midnight.

Syntax

Timer

Remarks

In Microsoft Windows the **Timer** function returns fractional portions of a second. On the Macintosh, timer resolution is one second.

TimeSerial Function

Returns a **Variant** (**Date**) containing the time for a specific hour, minute, and second.

Syntax

TimeSerial(*hour, minute, second*)

The **TimeSerial** function syntax has these named arguments:

Part	Description
hour	Required; **Variant** (**Integer**). Number between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive, or a numeric expression.
minute	Required; **Variant** (**Integer**). Any numeric expression.
second	Required; **Variant** (**Integer**). Any numeric expression.

Remarks

To specify a time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should be in the normal range for the unit; that is, 0–23 for hours and 0–59 for minutes and seconds. However, you can also specify relative times for each argument using any numeric expression that represents some 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 (`-15`) six hours before noon (`12 - 6`), or 5:45:00 A.M.

```
TimeSerial(12 - 6, -15, 0)
```

When any argument exceeds the normal range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. If any single argument is outside the range -32,768 to 32,767, an error occurs. If the time specified by the three arguments causes the date to fall outside the acceptable range of dates, an error occurs.

Returns a **Variant** (**Date**) containing the time.

Syntax

TimeValue(*time***)**

The required *time* argument is normally a string expression representing a time from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. However, *time* can also be any expression that represents a time in that range. If *time*

contains Null, **Null** is returned.

Remarks

You can enter valid times using a 12-hour or 24-hour clock. For example, `"2:24PM"` and `"14:24"` are both valid *time* arguments.

If the *time* argument contains date information, **TimeValue** doesn't return it. However, if *time* includes invalid date information, an error occurs.

Type Conversion Functions

Each function coerces an expression to a specific data type.

Syntax

CBool(*expression***)**

CByte(*expression***)**

CCur(*expression***)**

CDate(*expression***)**

CDbl(*expression***)**

CDec(*expression***)**

CInt(*expression***)**

CLng(*expression***)**

CSng(*expression***)**

CStr(*expression***)**

CVar(*expression***)**

The required *expression* argument is any string expression or numeric expression.

Return Types

The function name determines the return type as shown in the following:

Function	Return Type	Range for *expression* argument
CBool	Boolean	Any valid **string** or numeric expression.
CByte	Byte	0 to 255.
CCur	Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807.
CDate	Date	Any valid date expression.
CDbl	Double	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.
CDec	Decimal	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	Integer	-32,768 to 32,767; fractions are rounded.
CLng	Long	-2,147,483,648 to 2,147,483,647; fractions are rounded.
CSng	Single	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
CStr	String	Returns for CStr depend on the *expression* argument.
CVar	Variant	Same range as **Double** for numerics. Same range as **String** for non-numerics.

Remarks

If the *expression* passed to the function is outside the range of the data type being converted to, an error occurs.

In general, you can document your code using the data-type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CCur** to force currency arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

You should use the data-type conversion functions instead of **Val** to provide internationally aware conversions from one data type to another. For example, when you use **CCur**, different decimal separators, different thousand separators, and various currency options are properly recognized depending on the locale setting of your computer.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if *date* can be converted to a date or time. **CDate** recognizes date literals and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of your system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

A **CVDate** function is also provided for compatibility with previous versions of Visual Basic. The syntax of the **CVDate** function is identical to the **CDate** function, however, **CVDate** returns a **Variant** whose subtype is **Date** instead of an actual **Date** type. Since there is now an intrinsic **Date** type, there is no further

need for **CVDate**. The same effect can be achieved by converting an expression to a **Date,** and then assigning it to a **Variant**. This technique is consistent with the conversion of all other intrinsic types to their equivalent **Variant** subtypes.

Note The **CDec** function does not return a discrete data type; instead, it always returns a **Variant** whose value has been converted to a **Decimal** subtype.

TypeName Function

Returns a **String** that provides information about a variable.

Syntax

TypeName(*varname***)**

The required *varname* argument is a Variant containing any variable except a variable of a user-defined type.

Remarks

The string returned by **TypeName** can be any one of the following:

String returned	Variable
object type	An object whose type is *objecttype*
Byte	Byte value
Integer	Integer
Long	Long integer
Single	Single-precision floating-point number
Double	Double-precision floating-point number
Currency	Currency value
Decimal	Decimal value
Date	Date value

String	String
Boolean	Boolean value
Error	An error value
Empty	Uninitialized
Null	No valid data
Object	An object
Unknown	An object whose type is unknown
Nothing	Object variable that doesn't refer to an object

If *varname* is an array, the returned string can be any one of the possible returned strings (or **Variant**) with empty parentheses appended. For example, if *varname* is an array of integers, **TypeName** returns "`Integer()`".

UBound Function

Returns a Long containing the largest available subscript for the indicated dimension of an array.

Syntax

UBound(*arrayname*[**,** *dimension*]**)**

The **UBound** function syntax has these parts:

Part	Description

arrayname	Required. Name of the array variable; follows standard variable naming conventions.
dimension	Optional; **Variant** (**Long**). Whole number indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If *dimension* is omitted, 1 is assumed.

Remarks

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 returns the following values for an array with these dimensions:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

Statement	Return Value
`UBound(A, 1)`	100
`UBound(A, 2)`	3
`UBound(A, 3)`	4

UCase Function

Returns a **Variant** (**String**) containing the specified string, converted to uppercase.

Syntax

UCase(*string*)

The required *string* argument is any valid string expression. If *string* contains Null, **Null** is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

Returns the numbers contained in a string as a numeric value of appropriate type.

Syntax

Val(*string***)**

The required *string* argument is any valid string expression.

Remarks

The **Val** function stops reading the string at the first character it can't recognize as part of a number. Symbols and characters that are often considered parts of numeric values, such as dollar signs and commas, are not recognized. However, the function recognizes the radix prefixes `&O` (for octal) and `&H` (for hexadecimal). Blanks, tabs, and linefeed characters are stripped from the argument.

The following returns the value 1615198:

```
Val("    1615 198th Street N.E.")
```

In the code below, **Val** returns the decimal value -1 for the hexadecimal value shown:

```
Val("&HFFFF")
```

Note The **Val** function recognizes only the period (**.**) as a valid decimal separator. When different decimal separators are used, as in international applications, use **CDbl** instead to convert a string to a number.

VarType Function

Returns an **Integer** indicating the subtype of a variable.

Syntax

VarType(*varname***)**

The required *varname* argument is a Variant containing any variable except a variable of a user-defined type.

Return Values

Constant	Value	Description
vbEmpty	0	Empty (uninitialized)
vbNull	1	Null (no valid data)
vbInteger	2	Integer
vbLong	3	Long integer
vbSingle	4	Single-precision floating-point number
vbDouble	5	Double-precision floating-point number
vbCurrency	6	Currency value
vbDate	7	Date value
vbString	8	String
vbObject	9	Object

vbError	10	Error value
vbBoolean	11	Boolean value
vbVariant	12	**Variant** (used only with arrays of variants)
vbDataObject	13	A data access object
vbDecimal	14	Decimal value
vbByte	17	Byte value
vbUserDefinedType	36	Variants that contain user-defined types
vbArray	8192	Array

Note These constants are specified by Visual Basic for Applications. The names can be used anywhere in your code in place of the actual values.

Remarks

The **VarType** function never returns the value for **vbArray** by itself. It is always added to some other value to indicate an array of a particular type. The constant **vbVariant** is only returned in conjunction with **vbArray** to indicate that the argument to the **VarType** function is an array of type **Variant**. For example, the value returned for an array of integers is calculated as **vbInteger** + **vbArray**, or 8194. If an object has a default property, **VarType (***object***)** returns the type of the object's default property.

Weekday Function

Returns a **Variant** (**Integer**) containing a whole number representing the day of the week.

Syntax

Weekday(*date*, [***firstdayofweek***])**

The **Weekday** function syntax has these named arguments:

Part	Description
date	Required. Variant, numeric expression, string expression, or any combination, that can represent a date. If ***date*** contains Null, **Null** is returned.
firstdayofweek	Optional. A constant that specifies the first day of the week. If not specified, **vbSunday** is assumed.

Settings

The ***firstdayofweek*** argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use the NLS API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Return Values

The **Weekday** function can return any of these values:

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday

vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Remarks

If the **Calendar** property setting is Gregorian, the returned integer represents the Gregorian day of the week for the date argument. If the calendar is Hijri, the returned integer represents the Hijri day of the week for the date argument. For Hijri dates, the argument number is any numeric expression that can represent a date and/or time from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

Description

Returns a string indicating the specified day of the week.

Syntax

WeekdayName(*weekday***,** *abbreviate***,** *firstdayofweek***)**

The **WeekdayName** function syntax has these parts:

Part	Description
weekday	Required. The numeric designation for the day of the week. Numeric value of each day depends on setting of the *firstdayofweek* setting.
abbreviate	Optional. **Boolean** value that indicates if the weekday name is to be abbreviated. If omitted, the default is **False**, which means that the weekday name is not abbreviated.
firstdayofweek	Optional. Numeric value indicating the first day of the week. See Settings section for values.

Settings

The *firstdayofweek* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Year Function

Returns a **Variant** (**Integer**) containing a whole number representing the year.

Syntax

Year(*date***)**

The required *date* argument is any Variant, numeric expression, string expression, or any combination, that can represent a date. If *date* contains Null, **Null** is returned.

Note If the **Calendar** property setting is Gregorian, the returned integer represents the Gregorian year for the date argument. If the calendar is Hijri, the returned integer represents the Hijri year for the date argument. For Hijri dates, the argument number is any numeric expression that can represent a date and/or time from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

Arithmetic Operators

^ Operator

* Operator

/ Operator

\ Operator

Mod Operator

+ Operator

- Operator

Concatenation Operators

& Operator

+ Operator

Conversion Functions

Asc Function

CBool Function

CByte Function

CCur Function

CDate Function

CDec Function

CDbl Function

Chr Function

CInt Function

CLng Function

CSng Function

CStr Function

CVar Function

CVErr Function

Format Function

Hex Function

Oct Function

Str Function

Val Function

Data
Type

Summary

The following table shows the supported data types, including storage sizes and ranges.

Data type	Storage size	Range
Byte	1 byte	0 to 255
Boolean	2 bytes	**True** or **False**
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
Double (double-precision floating-point)	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency (scaled integer)	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point;

		+/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/-0.0000000000000000000000000001
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any **Object** reference
String (variable-length)	10 bytes + string length	0 to approximately 2 billion
String (fixed-length)	Length of string	1 to approximately 65,400
Variant (with numbers)	16 bytes	Any numeric value up to the range of a **Double**
Variant (with characters)	22 bytes + string length	Same range as for variable-length **String**
User-defined (using **Type**)	Number required by elements	The range of each element is the same as the range of its data type.

Note Arrays of any data type require 20 bytes of memory plus 4 bytes for each array dimension plus the number of bytes occupied by the data itself. The memory occupied by the data can be calculated by multiplying the number of data elements by the size of each element. For example, the data in a single-dimension array consisting of 4 **Integer** data elements of 2 bytes each occupies 8 bytes. The 8 bytes required for the data plus the 24 bytes of overhead brings the total memory requirement for the array to 32 bytes.

A **Variant** containing an array requires 12 bytes more than the array alone.

Note Use the **StrConv** function to convert one type of string data to another.

The following is a list of nonintrinsic math functions that can be derived from the intrinsic math functions:

Function	Derived equivalents
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)) + 2 * Atn(1)
Inverse Secant	Arcsec(X) = Atn(X / Sqr(X * X – 1)) + Sgn((X) – 1) * (2 * Atn(1))
Inverse Cosecant	Arccosec(X) = Atn(X / Sqr(X * X - 1)) + (Sgn(X) – 1) * (2 * Atn(1))
Inverse Cotangent	Arccotan(X) = Atn(X) + 2 * Atn(1)
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))
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	HArctan(X) = Log((1 + X) / (1 – X)) / 2

Tangent	
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 to base N	LogN(X) = Log(X) / Log(N)

Logical Operators

And Operator

Eqv Operator

Imp Operator

Not Operator

Or Operator

Xor Operator

Math Functions

Abs Function

Atn Function

Cos Function

Exp Function

Fix Function

Int Function

Log Function

Rnd Function

Sgn Function

Sin Function

Sqr Function

Tan Function

Derived Math Functions

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence.

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. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

Arithmetic	Comparison	Logical
Exponentiation (^)	Equality (=)	**Not**
Negation (–)	Inequality (<>)	**And**
Multiplication and division (*, /)	Less than (<)	**Or**

Integer division (\)	Greater than (>)	**Xor**
Modulus arithmetic (**Mod**)	Less than or equal to (<=)	**Eqv**
Addition and subtraction (+, –)	Greater than or equal to (>=)	**Imp**
String concatenation (**&**)	**Like** **Is**	

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. When addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right. 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, operator precedence is maintained.

The string concatenation operator (**&**) is not an arithmetic operator, but in precedence, it does follow all arithmetic operators and precede all comparison operators.

The **Like** operator is equal in precedence to all comparison operators, but is actually a pattern-matching operator.

The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

Operator Summary

Operators	Description
Arithmetic Operators	Operators used to perform mathematical calculations.
Comparison Operators	Operators used to perform comparisons.
Concatenation Operators	Operators used to combine strings.
Logical Operators	Operators used to perform logical operations.

Action	Keywords
Verify an array.	IsArray
Create an array.	Array
Change default lower limit.	Option Base
Declare and initialize an array.	Dim, Private, Public, ReDim, Static
Find the limits of an array.	LBound, UBound
Reinitialize an array.	Erase, ReDim

Action	Keywords
Create a **Collection** object.	Collection
Add an object to a collection.	Add
Remove an object from a collection.	Remove
Reference an item in a collection.	Item

Compiler Directive Keyword Summary

Action	Keywords
Define compiler constant.	#Const
Compile selected blocks of code.	#If...Then...#Else

Action	Keywords
Branch.	GoSub...Return, GoTo, On Error, On...GoSub, On...GoTo
Exit or pause the program.	DoEvents, End, Exit, Stop
Loop.	Do...Loop, For...Next, For Each...Next, While...Wend, With
Make decisions.	Choose, If...Then...Else, Select Case, Switch
Use procedures.	Call, Function, Property Get, Property Let, Property Set, Sub

Action	Keywords
ANSI value to string.	Chr
String to lowercase or uppercase.	Format, LCase, Ucase
Date to serial number.	DateSerial, DateValue
Decimal number to other bases.	Hex, Oct
Number to string.	Format, Str
One data type to another.	CBool, CByte, CCur, CDate, CDbl, CDec, CInt, CLng, CSng, CStr, CVar, CVErr, Fix, Int
Date to day, month, weekday, or year.	Day, Month, Weekday, Year
Time to hour, minute, or second.	Hour, Minute, Second
String to ASCII value.	Asc
String to number.	Val
Time to serial number.	TimeSerial, TimeValue

Action	Keywords
Convert between data types.	CBool, CByte, CCur, CDate, CDbl, CDec, CInt, CLng, CSng, CStr, CVar, CVErr, Fix, Int
Set intrinsic data types.	Boolean, Byte, Currency, Date, Double, Integer, Long, Object, Single, String, Variant (default)
Verify data types.	IsArray, IsDate, IsEmpty, IsError, IsMissing, IsNull, IsNumeric, IsObject

Action	Keywords
Get the current date or time.	Date, Now, Time
Perform date calculations.	DateAdd, DateDiff, DatePart
Return a date.	DateSerial, DateValue
Return a time.	TimeSerial, TimeValue
Set the date or time.	Date, Time
Time a process.	Timer

Action	Keywords
Change directory or folder.	ChDir
Change the drive.	ChDrive
Copy a file.	FileCopy
Make directory or folder.	MkDir
Remove directory or folder.	RmDir
Rename a file, directory, or folder.	Name
Return current path.	CurDir
Return file date/time stamp.	FileDateTime
Return file, directory, label attributes.	GetAttr
Return file length.	FileLen
Return file name or volume label.	Dir
Set attribute information for a file.	SetAttr

Action	Keywords
Generate run-time errors.	Clear, Error, Raise
Get error messages.	Error
Provide error information.	Err
Return **Error** variant.	CVErr
Trap errors during run time.	On Error, Resume
Type verification.	IsError

Action	Keywords
Calculate depreciation.	DDB, SLN, SYD
Calculate future value.	FV
Calculate interest rate.	Rate
Calculate internal rate of return.	IRR, MIRR
Calculate number of periods.	NPer
Calculate payments.	IPmt, Pmt, PPmt
Calculate present value.	NPV, PV

Action	Keywords
Access or create a file.	Open
Close files.	Close, Reset
Control output appearance.	Format, Print, Print #, Spc, Tab, Width #
Copy a file.	FileCopy
Get information about a file.	EOF, FileAttr, FileDateTime, FileLen, FreeFile, GetAttr, Loc, LOF, Seek
Manage files.	Dir, Kill, Lock, Unlock, Name
Read from a file.	Get, Input, Input #, Line Input #
Return length of a file.	FileLen
Set or get file attributes.	FileAttr, GetAttr, SetAttr
Set read-write position in a file.	Seek
Write to a file.	Print #, Put, Write #

Category	Description
Arrays	Creating, defining, and using arrays.
Compiler Directives	Controlling compiler behavior.
Control Flow	Looping and controlling procedure flow.
Conversion	Converting numbers and data types.
Data Types	Data types and variant subtypes.
Dates and Times	Converting and using date and time expressions.
Directories and Files	Controlling the file system and processing files.
Errors	Trapping and returning error values.
Financial	Performing financial calculations.
Input and Output	Receiving input and displaying or printing output.
Math	Performing trigonometric and other mathematical calculations.
Miscellaneous	Starting other applications and processing events.
Operators	Comparing expressions and other operations.
String Manipulation	Manipulating strings and string type data.
Variables and Constants	Declaring and defining variables and constants.

Action	Keywords
Derive trigonometric functions.	Atn, Cos, Sin, Tan
General calculations.	Exp, Log, Sqr
Generate random numbers.	Randomize, Rnd
Get absolute value.	Abs
Get the sign of an expression.	Sgn
Perform numeric conversions.	Fix, Int

Action	Keywords
Process pending events.	DoEvents
Run other programs.	AppActivate, Shell
Send keystrokes to an application.	SendKeys
Sound a beep from computer.	Beep
System.	Environ
Provide a command-line string.	Command
Automation.	CreateObject, GetObject
Color.	QBColor, RGB

Operators Keyword Summary

Action	Keywords
Arithmetic.	^, –, *, /, \, Mod, +, &, =
Comparison.	=, <>, <, >, <=, >=, Like, Is
Logical operations.	Not, And, Or, Xor, Eqv, Imp

Action	Keywords
Delete program settings.	DeleteSetting
Read program settings.	GetSetting, GetAllSettings
Save program settings.	SaveSetting

Action	Keywords
Compare two strings.	StrComp
Convert strings.	StrConv
Convert to lowercase or uppercase.	Format, Lcase, Ucase
Create string of repeating character.	Space, String
Find length of a string.	Len
Format a string.	Format
Justify a string.	LSet, Rset
Manipulate strings.	InStr, Left, LTrim, Mid, Right, RTrim, Trim
Set string comparison rules.	Option Compare
Work with ASCII and ANSI values.	Asc, Chr

Action	Keywords
Assign value.	Let
Declare variables or constants.	Const, Dim, Private, Public, New, Static
Declare module as private.	Option Private Module
Get information about a variant.	IsArray, IsDate, IsEmpty, IsError, IsMissing, IsNull, IsNumeric, IsObject, TypeName, VarType
Refer to current object.	Me
Require explicit variable declarations.	Option Explicit
Set default data type.	Deftype

As

The **As** keyword is used in these contexts:

Const Statement

Declare Statement

Dim Statement

Function Statement

Name Statement

Open Statement

Private Statement

Property Get Statement

Property Let Statement

Property Set Statement

Public Statement

ReDim Statement

Static Statement

Sub Statement

Type Statement

The **Binary** keyword is used in these contexts:

Open Statement

Option Compare Statement

The **ByRef** keyword is used in these contexts:

Call Statement

Declare Statement

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Sub Statement

The **ByVal** keyword is used in these contexts:

Call Statement

Declare Statement

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Sub Statement

The **Date** keyword is used in these contexts:

Date Data Type

Date Function

Date Statement

Else

The **Else** keyword is used in these contexts:

If...Then...Else Statement

Select Case Statement

Empty

The **Empty** keyword is used as a Variant subtype. It indicates an uninitialized variable value.

The **Error** keyword is used in these contexts:

Error Function

Error Statement

On Error Statement

False

The **False** keyword has a value equal to 0.

For

The **For** keyword is used in these contexts:

For...Next Statement

For Each...Next Statement

Open Statement

Get

The **Get** keyword is used in these contexts:

Get Statement

Property Get Statement

The **Input** keyword is used in these contexts:

Input Function

Input # Statement

Line Input # Statement

Open Statement

Is

The **Is** keyword is used in these contexts:

If...Then...Else Statement

Is Operator

Select Case Statement

Len

The **Len** keyword is used in these contexts:

Len Function

Open Statement

Let

The **Let** keyword is used in these contexts:

Let Statement

Property Let Statement

The **Lock** keyword is used in these contexts:

Lock, Unlock Statements

Open Statement

Me

The **Me** keyword behaves like an implicitly declared variable. It is automatically available to every procedure in a class module. When a class can have more than

one instance, **Me** provides a way to refer to the specific instance of the class where the code is executing. Using **Me** is particularly useful for passing information about the currently executing instance of a class to a procedure in another module. For example, suppose you have the following procedure in a module:

```
Sub ChangeFormColor(FormName As Form)
    FormName.BackColor = RGB(Rnd * 256, Rnd * 2
End Sub
```

You can call this procedure and pass the current instance of the Form class as an argument using the following statement:

```
ChangeFormColor Me
```

The **Mid** keyword is used in these contexts:

Mid Function

Mid Statement

New

The **New** keyword is used in these contexts:

Dim Statement

Private Statement

Public Statement

Set Statement

Static Statement

Next

The **Next** keyword is used in these contexts:

For...Next Statement

For Each...Next Statement

On Error Statement

Resume Statement

Nothing

The **Nothing** keyword is used to disassociate an object variable from an actual object. Use the **Set** statement to assign **Nothing** to an object variable. For example:

```
Set MyObject = Nothing
```

Several object variables can refer to the same actual object. When **Nothing** is assigned to an object variable, that variable no longer refers to an actual object. When several object variables refer to the same object, memory and system resources associated with the object to which the variables refer are released only after all of them have been set to **Nothing**, either explicitly using **Set**, or implicitly after the last object variable set to **Nothing** goes out of scope.

Null

The Null keyword is used as a Variant subtype. It indicates that a variable contains no valid data.

The **On** keyword is used in these contexts:

On Error Statement

On...GoSub Statement

On...GoTo Statement

The **Option** keyword is used in these contexts:

Option Base Statement

Option Compare Statement

Option Explicit Statement

Option Private Statement

The **Optional** keyword is used in these contexts:

Declare Statement

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Sub Statement

The **ParamArray** keyword is used in these contexts:

Declare Statement

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Sub Statement

The **Print** keyword is used in these contexts:

Print Method

Print # Statement

Private

The **Private** keyword is used in these contexts:

Const Statement

Declare Statement

Enum Statement

Function Statement

Option Private Statement

Private Statement

Property Get Statement

Property Let Statement

Property Set Statement

Sub Statement

Type Statement

The **Property** keyword is used in these contexts:

Property Get Statement

Property Let Statement

Property Set Statement

Public

The **Public** keyword is used in these contexts:

Const Statement

Declare Statement

Enum Statement

Event Statement

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Public Statement

Sub Statement

Type Statement

The **Resume** keyword is used in these contexts:

On Error Statement

Resume Statement

The **Seek** keyword is used in these contexts:

Seek Function

Seek Statement

Set

The **Set** keyword is used in these contexts:

Set Statement

Property Set Statement

Static

The **Static** keyword is used in these contexts:

Function Statement

Property Get Statement

Property Let Statement

Property Set Statement

Static Statement

Sub Statement

The **Step** keyword is used in these contexts:

For...Next Statement

For Each...Next Statement

The **String** keyword is used in these contexts:

String Data Type

String Function

Then

The **Then** keyword is used in these contexts:

#If...Then...#Else Directive

If...Then...Else Statement

The **Time** keyword is used in these contexts:

Time Function

Time Statement

To

The **To** keyword is used in these contexts:

Dim Statement

For...Next Statement

Lock, Unlock Statements

Private Statement

Public Statement

ReDim Statement

Select Case Statement

Static Statement

Type Statement

True

The **True** keyword has a value equal to -1.

The **WithEvents** keyword is used in these contexts:

Dim Statement

Private Statement

Public Statement

Add Method

Adds a member to a **Collection** object.

Syntax

object.**Add** ***item, key, before, after***

The **Add** method syntax has the following object qualifier and named arguments:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list.
item	Required. An expression of any type that specifies the member to add to the collection.
key	Optional. A unique string expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection.
before	Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the ***before*** argument. If a numeric expression, ***before*** must be a number from 1 to the value of the collection's **Count** property. If a string expression, ***before*** must correspond to the ***key*** specified when the member being referred to was added to the collection. You can specify a ***before*** position or an ***after*** position, but not both.
after	Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the ***after*** argument. If numeric, ***after*** must be a number from 1 to the value of the collection's **Count** property. If

a string, ***after*** must correspond to the ***key*** specified when the member referred to was added to the collection. You can specify a ***before*** position or an ***after*** position, but not both.

Remarks

Whether the ***before*** or ***after*** argument is a string expression or numeric expression, it must refer to an existing member of the collection, or an error occurs.

An error also occurs if a specified ***key*** duplicates the ***key*** for an existing member of the collection.

Description

Adds a key and item pair to a **Dictionary** object.

Syntax

object.**Add** *key*, *item*

The **Add** method has the following parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
key	Required. The key associated with the item being added.
item	Required. The item associated with the key being added.

Remarks

An error occurs if the *key* already exists.

Description

Adds a new **Folder** to a **Folders** collection.

Syntax

object.**Add** *folderName*

The **Add** method has the following parts:

Part	Description
object	Required. Always the name of a **Folders** collection.
folderName	Required. The name of the new **Folder** being added.

Remarks

An error occurs if the *folderName* already exists.

Description

Appends a name to an existing path.

Syntax

object.**BuildPath(***path*, *name***)**

The **BuildPath** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
path	Required. Existing path to which *name* is appended. Path can be absolute or relative and need not specify an existing folder.
name	Required. Name being appended to the existing *path*.

Remarks

The **BuildPath** method inserts an additional path separator between the existing path and the name, only if necessary.

Clear Method

Clears all property settings of the **Err** object.

Syntax

object.**Clear**

The *object* is always the **Err** object.

Remarks

Use **Clear** to explicitly clear the **Err** object after an error has been handled, for example, when you use deferred error handling with **On Error Resume Next**. The **Clear** method is called automatically whenever any of the following statements is executed:

Any type of **Resume** statement

Exit Sub, **Exit Function**, **Exit Property**

Any **On Error** statement

Note The **On Error Resume Next** construct may be preferable to **On Error GoTo** when handling errors generated during access to other objects. Checking **Err** after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in **Err.Number**, as well as which object originally generated the error (the object specified in **Err.Source**).

Description

Closes an open **TextStream** file.

Syntax

object.**Close**

The *object* is always the name of a **TextStream** object.

Description

Copies a specified file or folder from one location to another.

Syntax

object.**Copy** *destination*[, *overwrite*]

The **Copy** method syntax has these parts:

Part	Description
object	Required. Always the name of a **File** or **Folder** object.
destination	Required. Destination where the file or folder is to be copied. Wildcard characters are not allowed.
overwrite	Optional. **Boolean** value that is **True** (default) if existing files or folders are to be overwritten; **False** if they are not.

Remarks

The results of the **Copy** method on a **File** or **Folder** are identical to operations performed using **FileSystemObject.CopyFile** or **FileSystemObject.CopyFolder** where the file or folder referred to by *object* is

passed as an argument. You should note, however, that the alternative methods are capable of copying multiple files or folders.

Description

Copies one or more files from one location to another.

Syntax

object.**CopyFile** *source*, *destination*[, *overwrite*]

The **CopyFile** method syntax has these parts:

Part	Description
object	Required. The *object* is always the name of a **FileSystemObject**.
source	Required. Character string file specification, which can include wildcard characters, for one or more files to be copied.
destination	Required. Character string destination where the file or files from *source* are to be copied. Wildcard characters are not allowed.
overwrite	Optional. **Boolean** value that indicates if existing files are to be overwritten. If **True**, files are overwritten; if **False**, they are not. The default is **True**. Note that **CopyFile** will

fail if *destination* has the read-only attribute set, regardless of the value of *overwrite*.

Remarks

Wildcard characters can only be used in the last path component of the *source* argument. For example, you can use:

```
FileSystemObject.CopyFile "c:\mydocuments\lette
```

But you can't use:

```
FileSystemObject.CopyFile "c:\mydocuments\*\R1?
```

If *source* contains wildcard characters or *destination* ends with a path separator (\), it is assumed that *destination* is an existing folder in which to copy matching files. Otherwise, *destination* is assumed to be the name of a file to create. In either case, three things can happen when an individual file is copied.

If *destination* does not exist, *source* gets copied. This is the usual case.

If *destination* is an existing file, an error occurs if *overwrite* is **False**. Otherwise, an attempt is made to copy *source* over the existing file.

If *destination* is a directory, an error occurs.

An error also occurs if a *source* using wildcard characters doesn't match any files. The **CopyFile** method stops on the first error it encounters. No attempt is made to roll back or undo any changes made before an error occurs.

Description

Recursively copies a folder from one location to another.

Syntax

object.**CopyFolder** *source*, *destination*[, over*r*write]

The **CopyFolder** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
source	Required. Character string folder specification, which can include wildcard characters, for one or more folders to be copied.
destination	Required. Character string destination where the folder and subfolders from *source* are to be copied. Wildcard characters are not allowed.
overwrite	Optional. **Boolean** value that indicates if existing folders are to be overwritten. If **True**, files are overwritten; if **False**, they are not. The default is **True**.

Remarks

Wildcard characters can only be used in the last path component of the *source* argument. For example, you can use:

```
FileSystemObject.CopyFolder "c:\mydocuments\let
```

But you can't use:

```
FileSystemObject.CopyFolder "c:\mydocuments\*\*
```

If *source* contains wildcard characters or *destination* ends with a path separator (\), it is assumed that *destination* is an existing folder in which to copy matching folders and subfolders. Otherwise, *destination* is assumed to be the name of a folder to create. In either case, four things can happen when an individual folder is copied.

If *destination* does not exist, the *source* folder and all its contents gets copied. This is the usual case.

If *destination* is an existing file, an error occurs.

If *destination* is a directory, an attempt is made to copy the folder and all its contents. If a file contained in *source* already exists in *destination*, an error occurs if *overwrite* is **False**. Otherwise, it will attempt to copy the file over the existing file.

If *destination* is a read-only directory, an error occurs if an attempt is made to copy an existing read-only file into that directory and *overwrite* is **False**.

An error also occurs if a *source* using wildcard characters doesn't match any folders.

The **CopyFolder** method stops on the first error it encounters. No attempt is made to roll back any changes made before an error occurs.

Description

Creates a folder.

Syntax

object.**CreateFolder(***foldername***)**

The **CreateFolder** method has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
foldername	Required. String expression that identifies the folder to create.

Remarks

An error occurs if the specified folder already exists.

Description

Creates a specified file name and returns a **TextStream** object that can be used to read from or write to the file.

Syntax

object.**CreateTextFile(***filename*[**,** *overwrite*[**,** *unicode*]]**)**

The **CreateTextFile** method has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject** or **Folder** object.
filename	Required. String expression that identifies the file to create.

overwrite	Optional. **Boolean** value that indicates if an existing file can be overwritten. The value is **True** if the file can be overwritten; **False** if it can't be overwritten. If omitted, existing files are not overwritten.
unicode	Optional. **Boolean** value that indicates whether the file is created as a Unicode or ASCII file. The value is **True** if the file is created as a Unicode file; **False** if it's created as an ASCII file. If omitted, an ASCII file is assumed.

Remarks

The following code illustrates how to use the **CreateTextFile** method to create and open a text file:

```
Sub CreateAfile
    Set fs = CreateObject("Scripting.FileSystem
    Set a = fs.CreateTextFile("c:\testfile.txt"
    a.WriteLine("This is a test.")
    a.Close
End Sub
```

If the *overwrite* argument is **False**, or is not provided, for a *filename* that already exists, an error occurs.

Description

Deletes a specified file or folder.

Syntax

object.**Delete** *force*

The **Delete** method syntax has these parts:

Part	Description
object	Required. Always the name of a **File** or **Folder** object.
force	Optional. **Boolean** value that is **True** if files or folders with the read-only attribute set are to be deleted; **False** (default) if they are not.

Remarks

An error occurs if the specified file or folder does not exist.

The results of the **Delete** method on a **File** or **Folder** are identical to operations performed using **FileSystemObject.DeleteFile** or

FileSystemObject.DeleteFolder.

The **Delete** method does not distinguish between folders that have contents and those that do not. The specified folder is deleted regardless of whether or not it has contents.

Description

Deletes a specified file.

Syntax

object.**DeleteFile** *filespec*[, *force*]

The **DeleteFile** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
filespec	Required. The name of the file to delete. The *filespec* can contain wildcard characters in the last path component.
force	Optional. **Boolean** value that is **True** if files with the read-only attribute set are to be deleted; **False** (default) if they are not.

Remarks

An error occurs if no matching files are found. The **DeleteFile** method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.

Description

Deletes a specified folder and its contents.

Syntax

object.**DeleteFolder** *folderspec*[, *force*]

The **DeleteFolder** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
folderspec	Required. The name of the folder to delete. The *folderspec* can contain wildcard characters in the last path component.
force	Optional. **Boolean** value that is **True** if folders with the read-only attribute set are to be deleted; **False** (default) if they are not.

Remarks

The **DeleteFolder** method does not distinguish between folders that have contents and those that do not. The specified folder is deleted regardless of whether or not it has contents.

An error occurs if no matching folders are found. The **DeleteFolder** method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.

Description

Returns **True** if the specified drive exists; **False** if it does not.

Syntax

object.**DriveExists(***drivespec***)**

The **DriveExists** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
drivespec	Required. A drive letter or a complete path specification.

Remarks

For drives with removable media, the **DriveExists** method returns **True** even if there are no media present. Use the **IsReady** property of the **Drive** object to determine if a drive is ready.

Description

Returns **True** if a specified key exists in the **Dictionary** object; **False** if it does not.

Syntax

object.**Exists(***key***)**

The **Exists** method syntax has these parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
key	Required. *Key* value being searched for in the **Dictionary** object.

Description

Returns **True** if a specified file exists; **False** if it does not.

Syntax

object.**FileExists(***filespec***)**

The **FileExists** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
filespec	Required. The name of the file whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the file isn't expected to exist in the current folder.

Description

Returns **True** if a specified folder exists; **False** if it does not.

Syntax

object.**FolderExists(***folderspec*)

The **FolderExists** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
folderspec	Required. The name of the folder whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the folder isn't expected to exist in the current folder.

Description

Returns a complete and unambiguous path from a provided path specification.

Syntax

object.**GetAbsolutePathName(***pathspec***)**

The **GetAbsolutePathName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
pathspec	Required. Path specification to change to a complete and unambiguous path.

Remarks

A path is complete and unambiguous if it provides a complete reference from the root of the specified drive. A complete path can only end with a path separator character (\) if it specifies the root folder of a mapped drive.

Assuming the current directory is c:\mydocuments\reports, the following table

illustrates the behavior of the **GetAbsolutePathName** method.

pathspec	**Returned path**
"c:"	"c:\mydocuments\reports"
"c:.."	"c:\mydocuments"
"c:\\\"	"c:\"
"c:*.*\may97"	"c:\mydocuments\reports*.*\may97"
"region1"	"c:\mydocuments\reports\region1"
"c:\..\..\mydocuments"	"c:\mydocuments"

Description

Returns a string containing the base name of the last component, less any file extension, in a path.

Syntax

object.**GetBaseName(***path***)**

The **GetBaseName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
path	Required. The path specification for the component whose base name is to be returned.

Remarks

The **GetBaseName** method returns a zero-length string ("") if no component matches the *path* argument.

Note The **GetBaseName** method works only on the provided *path* string. It

does not attempt to resolve the path, nor does it check for the existence of the specified path.

Description

Returns a **Drive** object corresponding to the drive in a specified path.

Syntax

object.**GetDrive** *drivespec*

The **GetDrive** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
drivespec	Required. The *drivespec* argument can be a drive letter (c), a drive letter with a colon appended (c:), a drive letter with a colon and path separator appended (c:\), or any network share specification (\\computer2\share1).

Remarks

For network shares, a check is made to ensure that the share exists.

An error occurs if *drivespec* does not conform to one of the accepted forms or

does not exist.

To call the **GetDrive** method on a normal path string, use the following sequence to get a string that is suitable for use as *drivespec*:

```
DriveSpec = GetDriveName(GetAbsolutePathName(Pa
```

Description

Returns a string containing the name of the drive for a specified path.

Syntax

object.**GetDriveName(***path***)**

The **GetDriveName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
path	Required. The path specification for the component whose drive name is to be returned.

Remarks

The **GetDriveName** method returns a zero-length string ("") if the drive can't be determined.

Note The **GetDriveName** method works only on the provided *path* string. It does not attempt to resolve the path, nor does it check for the existence of the

specified path.

Description

Returns a string containing the extension name for the last component in a path.

Syntax

object.**GetExtensionName(***path***)**

The **GetExtensionName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
path	Required. The path specification for the component whose extension name is to be returned.

Remarks

For network drives, the root directory (\) is considered to be a component.

The **GetExtensionName** method returns a zero-length string ("") if no component matches the *path* argument.

Description

Returns a **File** object corresponding to the file in a specified path.

Syntax

object.**GetFile(***filespec***)**

The **GetFile** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
filespec	Required. The *filespec* is the path (absolute or relative) to a specific file.

Remarks

An error occurs if the specified file does not exist.

Description

Returns the last component of specified path that is not part of the drive specification.

Syntax

object.**GetFileName(***pathspec***)**

The **GetFileName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
pathspec	Required. The path (absolute or relative) to a specific file.

Remarks

The **GetFileName** method returns a zero-length string ("") if *pathspec* does not end with the named component.

Note The **GetFileName** method works only on the provided path string. It does

not attempt to resolve the path, nor does it check for the existence of the specified path.

Description

Returns a **Folder** object corresponding to the folder in a specified path.

Syntax

object.**GetFolder(***folderspec***)**

The **GetFolder** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
folderspec	Required. The *folderspec* is the path (absolute or relative) to a specific folder.

Remarks

An error occurs if the specified folder does not exist.

Description

Returns a string containing the name of the parent folder of the last component in a specified path.

Syntax

object.**GetParentFolderName(***path***)**

The **GetParentFolderName** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
path	Required. The path specification for the component whose parent folder name is to be returned.

Remarks

The **GetParentFolderName** method returns a zero-length string ("") if there is no parent folder for the component specified in the *path* argument.

Note The **GetParentFolderName** method works only on the provided *path*

string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.

Description

Returns the special folder specified.

Syntax

object.**GetSpecialFolder(***folderspec***)**

The **GetSpecialFolder** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
folderspec	Required. The name of the special folder to be returned. Can be any of the constants shown in the Settings section.

Settings

The *folderspec* argument can have any of the following values:

Constant	Value	Description
WindowsFolder	0	The Windows folder contains files installed by the Windows operating system.

SystemFolder	1	The System folder contains libraries, fonts, and device drivers.
TemporaryFolder	2	The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.

Description

Returns a randomly generated temporary file or folder name that is useful for performing operations that require a temporary file or folder.

Syntax

object.**GetTempName**

The optional *object* is always the name of a **FileSystemObject**.

Remarks

The **GetTempName** method does not create a file. It provides only a temporary file name that can be used with **CreateTextFile** to create a file.

Item Method

Returns a specific member of a **Collection** object either by position or by key.

Syntax

object.**Item(***index***)**

The **Item** method syntax has the following object qualifier and part:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list.
index	Required. An expression that specifies the position of a member of the collection. If a numeric expression, *index* must be a number from 1 to the value of the collection's **Count** property. If a string expression, *index* must correspond to the ***key*** argument specified when the member referred to was added to the collection.

Remarks

If the value provided as *index* doesn't match any existing member of the collection, an error occurs.

The **Item** method is the default method for a collection. Therefore, the following lines of code are equivalent:

```
Print MyCollection(1)
Print MyCollection.Item(1)
```

Description

Returns an array containing all the items in a **Dictionary** object.

Syntax

object.**Items**

The *object* is always the name of a **Dictionary** object.

Remarks

The following code illustrates use of the **Items** method:

```
Dim a, d, i             'Create some variables
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
a = d.Items             'Get the items
For i = 0 To d.Count -1 'Iterate the array
    Print a(i)          'Print item
```

Next
...

Description

Returns an array containing all existing keys in a **Dictionary** object.

Syntax

object.**Keys**

The *object* is always the name of a **Dictionary** object.

Remarks

The following code illustrates use of the **Keys** method:

```
Dim a, d, i             'Create some variables
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
a = d.keys              'Get the keys
For i = 0 To d.Count -1 'Iterate the array
    Print a(i)          'Print key
```

Next
...

Description

Moves a specified file or folder from one location to another.

Syntax

object.**Move** *destination*

The **Move** method syntax has these parts:

Part	Description
object	Required. Always the name of a **File** or **Folder** object.
destination	Required. Destination where the file or folder is to be moved. Wildcard characters are not allowed.

Remarks

The results of the **Move** method on a **File** or **Folder** are identical to operations performed using **FileSystemObject.MoveFile** or **FileSystemObject.MoveFolder**. You should note, however, that the alternative methods are capable of moving multiple files or folders.

Description

Moves one or more files from one location to another.

Syntax

object.**MoveFile** *source*, *destination*

The **MoveFile** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
source	Required. The path to the file or files to be moved. The *source* argument string can contain wildcard characters in the last path component only.
destination	Required. The path where the file or files are to be moved. The *destination* argument can't contain wildcard characters.

Remarks

If *source* contains wildcards or *destination* ends with a path separator (\), it is assumed that *destination* specifies an existing folder in which to move the

matching files. Otherwise, *destination* is assumed to be the name of a destination file to create. In either case, three things can happen when an individual file is moved:

If *destination* does not exist, the file gets moved. This is the usual case.

If *destination* is an existing file, an error occurs.

If dest*i*nation is a directory, an error occurs.

An error also occurs if a wildcard character that is used in *source* doesn't match any files. The **MoveFile** method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs.

Important This method allows moving files between volumes only if supported by the operating system.

Description

Moves one or more folders from one location to another.

Syntax

object.**MoveFolder** *source*, *destination*

The **MoveFolder** method syntax has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
source	Required. The path to the folder or folders to be moved. The *source* argument string can contain wildcard characters in the last path component only.
destination	Required. The path where the folder or folders are to be moved. The *destination* argument can't contain wildcard characters.

Remarks

If *source* contains wildcards or *destination* ends with a path separator (\), it is assumed that *destination* specifies an existing folder in which to move the

matching files. Otherwise, *destination* is assumed to be the name of a destination folder to create. In either case, three things can happen when an individual folder is moved:

If *destination* does not exist, the folder gets moved. This is the usual case.

If *destination* is an existing file, an error occurs.

If *destination* is a directory, an error occurs.

An error also occurs if a wildcard character that is used in *source* doesn't match any folders. The **MoveFolder** method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs.

Important This method allows moving folders between volumes only if supported by the operating system.

Description

Opens a specified file and returns a **TextStream** object that can be used to read from, write to, or append to the file.

Syntax

object.**OpenAsTextStream(**[*iomode*, [*format*]]**)**

The **OpenAsTextStream** method syntax has these parts:

Part	Description
object	Required. Always the name of a **File** object.
iomode	Optional. Indicates input/output mode. Can be one of three constants: **ForReading**, **ForWriting**, or **ForAppending**.
format	Optional. One of three **Tristate** values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.

Settings

The *iomode* argument can have any of the following settings:

Constant	Value	Description
ForReading	1	Open a file for reading only. You can't write to this file.
ForWriting	2	Open a file for writing. If a file with the same name exists, its previous contents are overwritten.
ForAppending	8	Open a file and write to the end of the file.

The *format* argument can have any of the following settings:

Constant	Value	Description
TristateUseDefault	–2	Opens the file using the system default.
TristateTrue	–1	Opens the file as Unicode.
TristateFalse	0	Opens the file as ASCII.

Remarks

The **OpenAsTextStream** method provides the same functionality as the **OpenTextFile** method of the **FileSystemObject**. In addition, the **OpenAsTextStream** method can be used to write to a file.

The following code illustrates the use of the **OpenAsTextStream** method:

```
Sub TextStreamTest
    Const ForReading = 1, ForWriting = 2, ForAp
    Const TristateUseDefault = -2, TristateTrue
    Dim fs, f, ts, s
    Set fs = CreateObject("Scripting.FileSystem
    fs.CreateTextFile "test1.txt"            'C
    Set f = fs.GetFile("test1.txt")
    Set ts = f.OpenAsTextStream(ForWriting, Tri
```

```
    ts.Write "Hello World"
    ts.Close
    Set ts = f.OpenAsTextStream(ForReading, Tri
    s = ts.ReadLine
    MsgBox s
    ts.Close
End Sub
```

Description

Opens a specified file and returns a **TextStream** object that can be used to read from or append to the file.

Syntax

object.**OpenTextFile(***filename*[**,** *iomode*[**,** *create*[**,** *format*]]]**)**

The **OpenTextFile** method has these parts:

Part	Description
object	Required. Always the name of a **FileSystemObject**.
filename	Required. String expression that identifies the file to open.

iomode	Optional. Indicates input/output mode. Can be one of two constants, either **ForReading** or **ForAppending**.
create	Optional. **Boolean** value that indicates whether a new file can be created if the specified *filename* doesn't exist. The value is **True** if a new file is created; **False** if it isn't created. The default is **False**.
format	Optional. One of three **Tristate** values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.

Settings

The *iomode* argument can have either of the following settings:

Constant	Value	Description
ForReading	1	Open a file for reading only. You can't write to this file.
ForAppending	8	Open a file and write to the end of the file.

The *format* argument can have any of the following settings:

Constant	Value	Description
TristateUseDefault	–2	Opens the file using the system default.
TristateTrue	–1	Opens the file as Unicode.
TristateFalse	0	Opens the file as ASCII.

Remarks

The following code illustrates the use of the **OpenTextFile** method to open a file for appending text:

```
Sub OpenTextFileTest
    Const ForReading = 1, ForWriting = 2, ForAp
```

```
    Dim fs, f
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.OpenTextFile("c:\testfile.txt",
    f.Write "Hello world!"
    f.Close
End Sub
```

Print Method

Prints text in the **Immediate** window.

Syntax

object.**Print** [*outputlist*]

The **Print** method syntax has the following object qualifier and part:

Part	Description
object	Optional. An object expression that evaluates to an object in the Applies To list.
outputlist	Optional. Expression or list of expressions to print. If omitted, a blank line is printed.

The *outputlist* argument has the following syntax and parts:

{**Spc(***n***)** | **Tab(***n***)**} *expression charpos*

Part	Description
Spc(*n***)**	Optional. Used to insert space characters in the output, where *n* is the number of space characters to insert.
Tab(*n***)**	Optional. Used to position the insertion point at an absolute column number where *n* is the column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone.
expression	Optional. Numeric expression or string expression to print.
charpos	Optional. Specifies the insertion point for the next character. Use a semicolon (**;**) to position the insertion point immediately following the last character displayed. Use **Tab(***n***)** to position the insertion point at an absolute column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone. If *charpos* is omitted, the next character is printed on the next line.

Remarks

Multiple expressions can be separated with either a space or a semicolon.

All data printed to the **Immediate** window is properly formatted using the

decimal separator for the locale settings specified for your system. The keywords are output in the appropriate language for the host application.

For Boolean data, either `True` or `False` is printed. The **True** and **False** keywords are translated according to the locale setting for the host application.

Date data is written using the standard short date format recognized by your system. When either the date or the time component is missing or zero, only the data provided is written.

Nothing is written if *outputlist* data is Empty. However, if *outputlist* data is Null, `Null` is output. The **Null** keyword is appropriately translated when it is output.

For error data, the output is written as `Error errorcode`. The **Error** keyword is appropriately translated when it is output.

The *object* is required if the method is used outside a module having a default display space. For example an error occurs if the method is called in a standard module without specifying an *object*, but if called in a form module, *outputlist* is displayed on the form.

Note Because the **Print** method typically prints with proportionally-spaced characters, 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 a "W", occupies more than one fixed-width column, and a narrow letter, such as an "i", occupies less. To allow for cases where wider than average characters are used, your tabular columns must be 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.

Raise Method

Generates a run-time error.

Syntax

object.**Raise** ***number, source, description, helpfile, helpcontext***

The **Raise** method has the following object qualifier and named arguments:

Argument	Description
object	Required. Always the **Err** object.
number	Required. Long integer that identifies the nature of the error. Visual Basic errors (both Visual Basic-defined and user-defined errors) are in the range 0–65535. The range 0–512 is reserved for system errors; the range 513–65535 is available for user-defined errors. When setting the **Number** property to your own error code in a class module, you add your error code number to the **vbObjectError** constant. For example, to generate the error number 513, assign **vbObjectError** + 513 to the **Number** property.
source	Optional. String expression naming the object or application that generated the error. When setting this property for an object, use the form *project.class*. If *source* is not specified, the programmatic ID of the current Visual Basic project is used.

description	Optional. String expression describing the error. If unspecified, the value in **Number** is examined. If it can be mapped to a Visual Basic run-time error code, the string that would be returned by the **Error** function is used as **Description**. If there is no Visual Basic error corresponding to **Number**, the "Application-defined or object-defined error" message is used.
helpfile	Optional. The fully qualified path to the Help file in which help on this error can be found. If unspecified, Visual Basic uses the fully qualified drive, path, and file name of the Visual Basic Help file.
helpcontext	Optional. The context ID identifying a topic within ***helpfile*** that provides help for the error. If omitted, the Visual Basic Help file context ID for the error corresponding to the **Number** property is used, if it exists.

Remarks

All of the arguments are optional except ***number***. If you use **Raise** without specifying some arguments, and the property settings of the **Err** object contain values that have not been cleared, those values serve as the values for your error.

Raise is used for generating run-time errors and can be used instead of the **Error** statement. **Raise** is useful for generating errors when writing class modules, because the **Err** object gives richer information than is possible if you generate errors with the **Error** statement. For example, with the **Raise** method, the source that generated the error can be specified in the **Source** property, online Help for the error can be referenced, and so on.

Description

Reads a specified number of characters from a **TextStream** file and returns the resulting string.

Syntax

object.**Read(***characters***)**

The **Read** method syntax has these parts:

Part	Description
object	Required. Always the name of a **TextStream**object.
characters	Required. Number of characters you want to read from the file.

Description

Reads an entire **TextStream** file and returns the resulting string.

Syntax

object.**ReadAll**

The *object* is always the name of a **TextStream** object.

Remarks

For large files, using the **ReadAll** method wastes memory resources. Other techniques should be used to input a file, such as reading a file line by line.

Description

Reads an entire line (up to, but not including, the newline character) from a **TextStream** file and returns the resulting string.

Syntax

object.**ReadLine**

The *object* argument is always the name of a **TextStream** object.

Remove Method

Removes a member from a **Collection** object.

Syntax

object.**Remove** *index*

The **Remove** method syntax has the following object qualifier and part:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list.
index	Required. An expression that specifies the position of a member of the collection. If a numeric expression, *index* must be a number from 1 to the value of the collection's **Count** property. If a string expression, *index* must correspond to the ***key*** argument specified when the member referred to was added to the collection.

Remarks

If the value provided as *index* doesn’t match an existing member of the collection, an error occurs.

Description

Removes a key, item pair from a **Dictionary** object.

Syntax

object.**Remove(***key***)**

The **Remove** method syntax has these parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
key	Required. *Key* associated with the key, item pair you want to remove from the **Dictionary** object.

Remarks

An error occurs if the specified key, item pair does not exist.

The following code illustrates use of the **Remove** method:

```
Dim a, d, i                'Create some variables
```

```
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
a = d.Remove()          'Remove second pair
```

Description

The **RemoveAll** method removes all key, item pairs from a **Dictionary** object.

Syntax

object.**RemoveAll**

The *object* is always the name of a **Dictionary** object.

Remarks

The following code illustrates use of the **RemoveAll** method:

```
Dim a, d, i             'Create some variables
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
a = d.RemoveAll         'Clear the dictionary
```

Description

Skips a specified number of characters when reading a **TextStream** file.

Syntax

object.**Skip(***characters***)**

The **Skip** method syntax has these parts:

Part	Description
object	Required. Always the name of a **TextStream** object.
characters	Required. Number of characters to skip when reading a file.

Remarks

Skipped characters are discarded.

Description

Skips the next line when reading a **TextStream** file.

Syntax

object.**SkipLine**

The *object* is always the name of a **TextStream** object.

Remarks

Skipping a line means reading and discarding all characters in a line up to and including the next newline character.

An error occurs if the file is not open for reading.

Description

Writes a specified string to a **TextStream** file.

Syntax

object.**Write(***string***)**

The **Write** method syntax has these parts:

Part	Description
object	Required. Always the name of a **TextStream** object.
string	Required. The text you want to write to the file.

Remarks

Specified strings are written to the file with no intervening spaces or characters between each string. Use the **WriteLine** method to write a newline character or a string that ends with a newline character.

Description

Writes a specified number of newline characters to a **TextStream** file.

Syntax

object.**WriteBlankLines(***lines***)**

The **WriteBlankLines** method syntax has these parts:

Part	Description
object	Required. Always the name of a **TextStream** object.
lines	Required. Number of newline characters you want to write to the file.

Description

Writes a specified string and newline character to a **TextStream** file.

Syntax

object.**WriteLine(**[*string*]**)**

The **WriteLine** method syntax has these parts:

Part	Description
object	Required. Always the name of a **TextStream** object.
string	Optional. The text you want to write to the file. If omitted, a newline character is written to the file.

Character Set (0 – 127)

0		32	[space]	64	@	96	`
1		33	!	65	A	97	a
2		34	"	66	B	98	b
3		35	#	67	C	99	c
4		36	$	68	D	100	d
5		37	%	69	E	101	e
6		38	&	70	F	102	f
7		39	'	71	G	103	g
8	* *	40	(	72	H	104	h
9	* *	41	)	73	I	105	i
10	* *	42	*	74	J	106	j
11		43	+	75	K	107	k
12		44	,	76	L	108	l
13	* *	45	-	77	M	109	m
14		46	.	78	N	110	n
15		47	/	79	O	111	o
16		48	0	80	P	112	p
17		49	1	81	Q	113	q
18		50	2	82	R	114	r
19		51	3	83	S	115	s
20		52	4	84	T	116	t
21		53	5	85	U	117	u
22		54	6	86	V	118	v
23		55	7	87	W	119	w

24	56	8	88	X	120	x
25	57	9	89	Y	121	y
26	58	:	90	Z	122	z
27	59	;	91	[	123	{
28	60	<	92	\	124	|
29	61	=	93	]	125	}
30	62	>	94	^	126	~
31	63	?	95	_	127	

These characters aren't supported by Microsoft Windows.

* *Values 8, 9, 10, and 13 convert to backspace, tab, linefeed, and carriage return characters, respectively. They have no graphical representation but, depending on the application, can affect the visual display of text.

Character Set (128 – 255)

128	€	160	[space]	192	À	224	à
129	€	161	¡	193	Á	225	á
130	€	162	¢	194	Â	226	â
131	€	163	£	195	Ã	227	ã
132	€	164	¤	196	Ä	228	ä
133	€	165	¥	197	Å	229	å
134	€	166	¦	198	Æ	230	æ
135	€	167	§	199	Ç	231	ç
136	€	168	¨	200	È	232	è
137	€	169	©	201	É	233	é
138	€	170	ª	202	Ê	234	ê
139	€	171	«	203	Ë	235	ë
140	€	172	¬	204	Ì	236	ì
141	€	173		205	Í	237	í
142	€	174	®	206	Î	238	î
143	€	175	¯	207	Ï	239	ï
144	€	176	°	208	Ð	240	ð
145	€	177	±	209	Ñ	241	ñ
146	€	178	²	210	Ò	242	ò
147	€	179	³	211	Ó	243	ó
148	€	180	´	212	Ô	244	ô
149	€	181	µ	213	Õ	245	õ
150	€	182	¶	214	Ö	246	ö
151	€	183	·	215	×	247	÷

152	€	184	¸	216	Ø	248	ø
153	€	185	¹	217	Ù	249	ù
154	€	186	º	218	Ú	250	ú
155	€	187	»	219	Û	251	û
156	€	188	¼	220	Ü	252	ü
157	€	189	½	221	Ý	253	ý
158	€	190	¾	222	Þ	254	þ
159	€	191	¿	223	ß	255	ÿ

€These characters aren't supported by Microsoft Windows.

The values in the table are the Windows default. However, values in the ANSI character set above 127 are determined by the code page specific to your operating system.

Visual Basic documentation uses the following typographic conventions.

Convention	Description
Sub, **If**, **ChDir**, **Print**, **True**, **Debug**	Words in bold with initial letter capitalized indicate language-specific keywords.
Setup	Words you are instructed to type appear in bold.
object, *varname*, *arglist*	Italic, lowercase letters indicate placeholders for information you supply.
pathname, ***filenumber***	Bold, italic, and lowercase letters indicate placeholders for arguments where you can use either positional or named-argument syntax.
[*expressionlist*]	In syntax, items inside 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 are also enclosed in brackets. For example: [{**This** | **OrThat**}]
ESC, ENTER	Words in capital letters indicate key names and key sequences.
ALT+F1, CTRL+R	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.

Code Conventions

The following code conventions are used:

Sample Code	Description
`MyString = "Hello, world!"`	This font is used for code, variables, and error message text.
`' This is a comment.`	An apostrophe (') introduces code comments.

`MyVar = "This is an " _`
`& "example" _`
`& " of how to continue code."`
	A space and an underscore (_) continue a line of code.

Collection Object

A **Collection** object is an ordered set of items that can be referred to as a unit.

Remarks

The **Collection** object provides a convenient way to refer to a related group of items as a single object. The items, or members, in a collection need only be related by the fact that they exist in the collection. Members of a collection don't have to share the same data type.

A collection can be created the same way other objects are created. For example:

```
Dim X As New Collection
```

Once a collection is created, members can be added using the **Add** method and removed using the **Remove** method. Specific members can be returned from the collection using the **Item** method, while the entire collection can be iterated using the **For Each...Next** statement.

Debug Object

The **Debug** object sends output to the **Immediate** window at run time.

Description

Object that stores data key, item pairs.

Syntax

Scripting.Dictionary

Remarks

A **Dictionary** object is the equivalent of a PERL associative array. Items, which can be any form of data, are stored in the array. Each item is associated with a unique key. The key is used to retrieve an individual item and is usually a integer or a string, but can be anything except an array.

The following code illustrates how to create a **Dictionary** object:

```
Dim d                       'Create a variable
Set d = CreateObject(Scripting.Dictionary)
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
```

Description

Provides access to the properties of a particular disk drive or network share.

Remarks

The following code illustrates the use of the **Drive** object to access drive properties:

```
Sub ShowFreeSpace(drvPath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(fs.GetDriveName(drvPath
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName  & vbCrLf
```

```
    s = s & "Free Space: " & FormatNumber(d.Fre
    s = s & " Kbytes"
    MsgBox s
End Sub
```

Description

Read-only collection of all available drives.

Remarks

Removable-media drives need not have media inserted for them to appear in the **Drives** collection.

The following code illustrates how to get the **Drives** collection and iterate the collection using the **For Each...Next** statement:

```
Sub ShowDriveList
    Dim fs, d, dc, s, n
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set dc = fs.Drives
    For Each d in dc
        s = s & d.DriveLetter & " - "
        If d.DriveType = Remote Then
            n = d.ShareName
        Else
            n = d.VolumeName
        End If
        s = s & n & vbCrLf
    Next
    MsgBox s
End Sub
```

Err Object

Contains information about run-time errors.

Remarks

The properties of the **Err** object are set by the generator of an error — Visual Basic, an object, or the programmer.

The default property of the **Err** object is **Number**. Because the default property can be represented by the object name **Err**, earlier code written using the **Err** function or **Err** statement doesn't have to be modified.

When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the **Raise** method.

The **Err** object's properties are reset to zero or zero-length strings ("") after an **Exit Sub**, **Exit Function**, **Exit Property** or **Resume Next** statement within an error-handling routine. Using any form of the **Resume** statement outside of an error-handling routine will not reset the **Err** object's properties. The **Clear** method can be used to explicitly reset **Err**.

Use the **Raise** method, rather than the **Error** statement, to generate run-time errors for system errors and class modules. Using the **Raise** method in other code depends on the richness of the information you want to return.

The **Err** object is an intrinsic object with global scope. There is no need to create an instance of it in your code.

Description

Provides access to all the properties of a file.

Remarks

The following code illustrates how to obtain a **File** object and how to view one of its properties.

```
Sub ShowFileInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = f.DateCreated
    MsgBox s
```

```
End Sub
```

Description

Collection of all **File** objects within a folder.

Remarks

The following code illustrates how to get a **Files** collection and iterate the collection using the **For Each...Next** statement:

```
Sub ShowFolderList(folderspec)
    Dim fs, f, f1, fc, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(folderspec)
    Set fc = f.Files
    For Each f1 in fc
```

```
        s = s & f1.name
        s = s & vbCrLf
    Next
    MsgBox s
End Sub
```

Description

Provides access to a computer's file system.

Syntax

Scripting.FileSystemObject

Remarks

The following code illustrates how the **FileSystemObject** is used to return a **TextStream** object that can be read from or written to:

```
Set fs = CreateObject("Scripting.FileSystemObje
Set a = fs.CreateTextFile("c:\testfile.txt", Tr
a.WriteLine("This is a test.")
```

```
a.Close
```

In the code shown above, the **CreateObject** function returns the **FileSystemObject** (`fs`). The **CreateTextFile** method then creates the file as a **TextStream** object (`a`), and the **WriteLine** method writes a line of text to the created text file. The **Close** method flushes the buffer and closes the file.

Description

Provides access to all the properties of a folder.

Remarks

The following code illustrates how to obtain a **Folder** object and how to return one of its properties:

```
Sub ShowFolderInfo(folderspec)
    Dim fs, f, s,
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(folderspec)
    s = f.DateCreated
    MsgBox s
```

```
End Sub
```

Description

Collection of all **Folder** objects contained within a **Folder** object.

Remarks

The following code illustrates how to get a **Folders** collection and how to iterate the collection using the **For Each...Next** statement:

```
Sub ShowFolderList(folderspec)
    Dim fs, f, f1, fc, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(folderspec)
    Set fc = f.SubFolders
    For Each f1 in fc
```

```
        s = s & f1.name
        s = s &  vbCrLf
    Next
    MsgBox s
End Sub
```

Description

Facilitates sequential access to file.

Syntax

TextStream.{*property* | *method*}

The *property* and *method* arguments can be any of the properties and methods associated with the **TextStream** object. Note that in actual usage **TextStream** is replaced by a variable placeholder representing the **TextStream** object returned from the **FileSystemObject**.

Remarks

In the following code, a is the **TextStream** object returned by the

CreateTextFile method on the **FileSystemObject**:

```
Set fs = CreateObject("Scripting.FileSystemObje
Set a = fs.CreateTextFile("c:\testfile.txt", Tr
a.WriteLine("This is a test.")
a.Close
```

WriteLine and **Close** are two methods of the **TextStream** Object.

& Operator

Used to force string concatenation of two expressions.

Syntax

result = *expression1* **&** *expression2*

The **&** operator syntax has these parts:

Part	Description
result	Required; any String or Variant variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If an *expression* is not a string, it is converted to a **String** variant. The data type of *result* is **String** if both expressions are string expressions; otherwise, *result* is a **String** variant. If both expressions are Null, *result* is **Null**. However, if only one *expression* is **Null**, that expression is treated as a zero-length string ("") when concatenated with the other expression. Any expression that is Empty is also treated as a zero-length string.

* Operator

Used to multiply two numbers.

Syntax

result = *number1***number2*

The ***** operator syntax has these parts:

Part	Description

result Required; any numeric variable.

number1 Required; any numeric expression.

number2 Required; any numeric expression.

Remarks

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

If	Then *result* is
Multiplication involves a **Single** and a **Long**,	converted to a **Double**.
The data type of *result* is a **Long**, **Single**, or **Date** variant that overflows its legal range,	converted to a **Variant** containing a **Double**.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.
the data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.

If one or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as 0.

Note The order of precision used by multiplication is not the same as the order of precision used by addition and subtraction.

+ Operator

Used to sum two numbers.

Syntax

result = *expression1*+*expression2*

The + operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

When you use the + operator, you may not be able to determine whether addition or string concatenation will occur. Use the **&** operator for concatenation to eliminate ambiguity and provide self-documenting code.

If at least one expression is not a Variant, the following rules apply:

If	Then
Both expressions are numeric data types (Byte, Boolean, Integer, Long, Single, Double, Date, Currency, or Decimal)	Add.
Both expressions are String	Concatenate.
One expression is a numeric data type and the other is any **Variant** except Null	Add.
One expression is a **String** and the other is any **Variant** except **Null**	Concatenate.

One expression is an Empty **Variant**	Return the remaining expression unchanged as *result*.
One expression is a numeric data type and the other is a **String**	A `Type mismatch` error occurs.
Either expression is **Null**	*result* is **Null**.

If both expressions are **Variant** expressions, the following rules apply:

If	Then
Both **Variant** expressions are numeric	Add.
Both **Variant** expressions are strings	Concatenate.
One **Variant** expression is numeric and the other is a string	Add.

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

If	Then *result* is
A **Single** and a **Long** are added,	a **Double**.
The data type of *result* is a **Long**, **Single**, or **Date** variant that overflows its legal range,	converted to a **Double** variant.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.
The data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.
A **Date** is added to any data type,	a **Date**.

If one or both expressions are **Null** expressions, *result* is **Null**. If both expressions are **Empty**, *result* is an **Integer**. However, if only one expression is **Empty**, the other expression is returned unchanged as *result*.

Note The order of precision used by addition and subtraction is not the same as the order of precision used by multiplication.

- Operator

Used to find the difference between two numbers or to indicate the negative value of a numeric expression.

Syntax 1

result = *number1*–*number2*

Syntax 2

–*number*

The – operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number	Required; any numeric expression.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

In Syntax 1, the – operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the – operator is used as the unary negation operator to indicate the negative value of an expression.

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

If	Then *result* is
Subtraction involves a **Single** and a **Long**,	converted to a **Double**.
The data type of *result* is a **Long**, **Single**, or Date variant that overflows its legal range,	converted to a Variant containing a **Double**.
The data type of *result* is a **Byte** variant that overflows its legal range,	converted to an **Integer** variant.
The data type of *result* is an **Integer** variant that overflows its legal range,	converted to a **Long** variant.
Subtraction involves a **Date** and any	a **Date**.

other data type,

Subtraction involves two **Date** expressions, a **Double**.

One or both expressions are Null expressions, *result* is **Null**. If an expression is Empty, it is treated as 0.

Note The order of precision used by addition and subtraction is not the same as the order of precision used by multiplication.

/ Operator

Used to divide two numbers and return a floating-point result.

Syntax

result = *number1*/*number2*

The / operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

The data type of *result* is usually a Double or a **Double** variant. The following

are exceptions to this rule:

If	Then *result* is
Both expressions are Byte, Integer, or Single expressions,	a **Single** unless it overflows its legal range; in which case, an error occurs.
Both expressions are **Byte**, **Integer**, or **Single** variants,	a **Single** variant unless it overflows its legal range; in which case, *result* is a Variant containing a **Double**.
Division involves a Decimal and any other data type,	a **Decimal** data type.

One or both expressions are Null expressions, *result* is **Null**. Any expression that is Empty is treated as 0.

\ Operator

Used to divide two numbers and return an integer result.

Syntax

result = *number1**number2*

The \ operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

Before division is performed, the numeric expressions are rounded to Byte, Integer, or Long expressions.

Usually, the data type of *result* is a **Byte**, **Byte** variant, **Integer**, **Integer** variant, **Long**, or **Long** variant, regardless of whether *result* is a whole number. Any fractional portion is truncated. However, if any expression is Null, *result* is **Null**. Any expression that is Empty is treated as 0.

^Operator

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

Syntax

result = *number*^*exponent*

The ^ operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number	Required; any numeric expression.
exponent	Required; any numeric expression.

Remarks

A *number* 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 containing a **Double**. However, if either *number* or *exponent* is a Null expression, *result* is **Null**.

= Operator

Description

Used to assign a value to a variable or property.

Syntax

variable = *value*

The = operator syntax has these parts:

Part	Description
variable	Any variable or any writable property.
value	Any numeric or string literal, constant, or expression.

Remarks

The name on the left side of the equal sign can be a simple scalar variable or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time.

And Operator

Used to perform a logical conjunction on two expressions.

Syntax

result = *expression1* **And** *expression2*

The **And** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If both expressions evaluate to **True**, *result* is **True**. If either expression evaluates to **False**, *result* is **False.** The following table illustrates how *result* is determined:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
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 bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	0
0	1	0

1	0	0
1	1	1

Comparison Operators

Used to compare expressions.

Syntax

result = *expression1 comparisonoperator expression2*

result = *object1* **Is** *object2*

result = *string* **Like** *pattern*

Comparison operators have these parts:

Part	Description
result	Required; any numeric variable.
expression	Required; any expression.
comparisonoperator	Required; any comparison operator.
object	Required; any object name.
string	Required; any string expression.
pattern	Required; any string expression or range of characters.

Remarks

The following table contains a list of the comparison operators and the

conditions that determine whether *result* is **True**, **False**, or Null:

Operator	True if	False if	Null if
< (Less than)	*expression1* < *expression2*	*expression1* >= *expression2*	*expression1* or *expression2* = **Null**
<= (Less than or equal to)	*expression1* <= *expression2*	*expression1* > *expression2*	*expression1* or *expression2* = **Null**
> (Greater than)	*expression1* > *expression2*	*expression1* <= *expression2*	*expression1* or *expression2* = **Null**
>= (Greater than or equal to)	*expression1* >= *expression2*	*expression1* < *expression2*	*expression1* or *expression2* = **Null**
= (Equal to)	*expression1* = *expression2*	*expression1* <> *expression2*	*expression1* or *expression2* = **Null**
<> (Not equal to)	*expression1* <> *expression2*	*expression1* = *expression2*	*expression1* or *expression2* = **Null**

Note The **Is** and **Like** operators have specific comparison functionality that differs from the operators in the table.

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 the result when either expression is not a Variant:

If	Then
Both expressions are numeric data types (Byte, Boolean, Integer, Long, Single, Double, Date, Currency, or Decimal)	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** that is, or can be, a number	Perform a numeric comparison.
One expression is a numeric data type and the other is a string **Variant** that	A `Type Mismatch` error occurs.

can't be converted to a number	
One expression is a **String** and the other is any **Variant** except a **Null**	Perform a string comparison.
One expression is Empty and the other is a numeric data type	Perform a numeric comparison, using 0 as the **Empty** expression.
One expression is **Empty** and the other is a **String**	Perform a string comparison, using a zero-length string ("") as the **Empty** expression.

If *expression1* and *expression2* are both **Variant** expressions, their underlying type determines how they are compared. The following table shows how the expressions are compared or the result from the comparison, depending on the underlying type of the **Variant**:

If	Then
Both **Variant** expressions are numeric	Perform a numeric comparison.
Both **Variant** expressions are strings	Perform a string comparison.
One **Variant** expression is numeric and the other is a string	The numeric expression is less than the string expression.
One **Variant** expression is **Empty** and the other is numeric	Perform a numeric comparison, using 0 as the **Empty** expression.
One **Variant** expression is **Empty** and the other is a string	Perform a string comparison, using a zero-length string ("") as the **Empty** expression.
Both **Variant** expressions are **Empty**	The expressions are equal.

When a **Single** is compared to a **Double**, the **Double** is rounded to the precision of the **Single**.

If a **Currency** is compared with a **Single** or **Double**, the **Single** or **Double** is converted to a **Currency**. Similarly, when a **Decimal** is compared with a **Single** or **Double**, the **Single** or **Double** is converted to a **Decimal**. For **Currency**, any fractional value less than .0001 may be lost; for **Decimal**, any fractional value

less than 1E-28 may be lost, or an overflow error can occur. Such fractional value loss may cause two values to compare as equal when they are not.

Eqv Operator

Used to perform a logical equivalence on two expressions.

Syntax

result = *expression1* **Eqv** *expression2*

The **Eqv** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

The **Eqv** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	1
0	1	0
1	0	0
1	1	1

Imp Operator

Used to perform a logical implication on two expressions.

Syntax

result = *expression1* **Imp** *expression2*

The **Imp** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

The following table illustrates how *result* is determined:

If *expression1* is	And *expression2* is	The *result* is
True	**True**	**True**
True	**False**	**False**
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 bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	The *result* is
0	0	1
0	1	1
1	0	0
1	1	1

Used to compare two object reference variables.

Syntax

result = *object1* **Is** *object2*

The **Is** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
object1	Required; any object name.
object2	Required; any object name.

Remarks

If *object1* and *object2* both refer to the same object, *result* is **True**; if they do not, *result* is **False**. Two variables can be made to refer to the same object in several ways.

In the following example, A has been set to refer to the same object as B:

```
Set A = B
```

The following example makes A and B refer to the same object as C:

```
Set A = C
Set B = C
```

Like Operator

Used to compare two strings.

Syntax

result = *string* **Like** *pattern*

The **Like** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
string	Required; any string expression.
pattern	Required; any string expression conforming to the pattern-matching conventions described in Remarks.

Remarks

If *string* matches *pattern*, *result* is **True**; if there is no match, *result* is **False**. If either *string* or *pattern* is Null, *result* is **Null**.

The behavior of the **Like** operator depends on the **Option Compare** statement. The default string-comparison method for each module is **Option Compare Binary**.

Option Compare Binary results in string comparisons based on a sort order

derived from the internal binary representations of the characters. Sort order is determined by the code page. In the following example, a typical binary sort order is shown:

A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø

Option Compare Text results in string comparisons based on a case-insensitive, textual sort order determined by your system's locale. When you sort The same characters using **Option Compare Text**, the following text sort order is produced:

(A=a) < (À=à) < (B=b) < (E=e) < (Ê=ê) < (Z=z) < (Ø=ø)

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to use wildcard characters, character lists, or character ranges, in any combination, to match strings. The following table shows the characters allowed in *pattern* and what they match:

Characters in *pattern*	**Matches in *string***
?	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 *string* and can include almost any character code, including digits.

Note To match the special characters left bracket (**[**), question mark (**?**), number sign (#), and asterisk (*****), enclose them in brackets. The right bracket (**]**) can't be used within a group to match itself, but it can be used outside a group as an individual character.

By using a hyphen (**–**) to separate the upper and lower bounds of the range,

charlist can specify a range of characters. For example, `[A-Z]` results in a match if the corresponding character position in *string* contains any uppercase letters in the range A–Z. Multiple ranges are included within the brackets without delimiters.

The meaning of a specified range depends on the character ordering valid at run time (as determined by **Option Compare** and the locale setting of the system the code is running on). Using the **Option Compare Binary** example, the range `[A-E]` matches A, B and E. With **Option Compare Text**, `[A-E]` matches A, a, À, à, B, b, E, e. The range does not match Ê or ê because accented characters fall after unaccented characters in the sort order.

Other important rules for pattern matching include the following:

An exclamation point (**!**) at the beginning of *charlist* means that a match is made if any character except the characters in *charlist* is found in *string*. When used outside brackets, the exclamation point matches itself.

A hyphen (**–**) can appear either at the beginning (after an exclamation point 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 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 considered a zero-length string ("").

In some languages, there are special characters in the alphabet that represent two separate characters. For example, several languages use the character "æ" to represent the characters "a" and "e" when they appear together. The **Like** operator recognizes that the single special character and the two individual characters are equivalent.

When a language that uses a special character is specified in the system locale settings, an occurrence of the single special character in either *pattern* or *string*

matches the equivalent 2-character sequence in the other string. Similarly, a single special character in *pattern* enclosed in brackets (by itself, in a list, or in a range) matches the equivalent 2-character sequence in *string*.

Mod Operator

Used to divide two numbers and return only the remainder.

Syntax

result = *number1* **Mod** *number2*

The **Mod** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
number1	Required; any numeric expression.
number2	Required; any numeric expression.

Remarks

The modulus, or remainder, operator divides *number1* by *number2* (rounding floating-point numbers to integers) and returns only the remainder as *result*. For example, in the following expression, A (*result*) equals 5.

```
A = 19 Mod 6.7
```

Usually, the data type of *result* is a Byte, **Byte** variant, Integer, **Integer** variant, Long, or Variant containing a **Long**, regardless of whether or not *result* is a whole number. Any fractional portion is truncated. However, if any expression is

Null, *result* is **Null**. Any expression that is Empty is treated as 0.

Not Operator

Used to perform logical negation on an expression.

Syntax

result = **Not** *expression*

The **Not** operator syntax has these parts:

Part	Description

result	Required; any numeric variable.
expression	Required; any expression.

Remarks

The following table illustrates how *result* is determined:

If *expression* is	**Then *result* is**
True	**False**
False	**True**
Null	**Null**

In addition, the **Not** operator inverts the bit values of any variable and sets the corresponding bit in *result* according to the following table:

If bit in *expression* is	**Then bit in *result* is**
o	1
1	0

Or Operator

Used to perform a logical disjunction on two expressions.

Syntax

result = *expression1* **Or** *expression2*

The **Or** operator syntax has these parts:

Part	Description
result	Required; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If either or both expressions evaluate to **True**, *result* is **True**. The following table illustrates how *result* is determined:

If *expression1* is	And *expression2* is	Then *result* is
True	**True**	**True**
True	**False**	**True**
True	Null	**True**
False	**True**	**True**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

The **Or** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	0
0	1	1
1	0	1
1	1	1

Xor Operator

Used to perform a logical exclusion on two expressions.

Syntax

[*result* =] *expression1* **Xor** *expression2*

The **Xor** operator syntax has these parts:

Part	Description
result	Optional; any numeric variable.
expression1	Required; any expression.
expression2	Required; any expression.

Remarks

If one, and only one, of the expressions evaluates to **True**, *result* is **True**. However, if either expression is Null, *result* is also **Null**. When neither expression is **Null**, *result* is determined according to the following table:

If *expression1* is	And *expression2* is	Then *result* is
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

The **Xor** operator performs as both a logical and bitwise operator. A bit-wise comparison of two expressions using exclusive-or logic to form the result, as shown in the following table:

If bit in *expression1* is	And bit in *expression2* is	Then *result* is
0	0	0
0	1	1
1	0	1
1	1	0

Description

Read-only property that returns **True** if the file pointer immediately precedes the end-of-line marker in a **TextStream** file; **False** if it does not.

Syntax

object.**AtEndOfLine**

The *object* is always the name of a **TextStream** object.

Remarks

The **AtEndOfLine** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.

The following code illustrates the use of the **AtEndOfLine** property:

```
Dim fs, a, retstring
Set fs = CreateObject("Scripting.FileSystemObje
Set a = fs.OpenTextFile("c:\testfile.txt", ForR
Do While a.AtEndOfLine <> True
```

```
    retstring = a.Read(1)
    ...
Loop
a.Close
```

Description

Read-only property that returns **True** if the file pointer is at the end of a **TextStream** file; **False** if it is not.

Syntax

object.**AtEndOfStream**

The *object* is always the name of a **TextStream** object.

Remarks

The **AtEndOfStream** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.

The following code illustrates the use of the **AtEndOfStream** property:

```
Dim fs, a, retstring
Set fs = CreateObject("Scripting.FileSystemObje
Set a = fs.OpenTextFile("c:\testfile.txt", ForR
Do While a.AtEndOfStream <> True
```

```
    retstring = a.ReadLine
    ...
Loop
a.Close
```

Description

Sets or returns the attributes of files or folders. Read/write or read-only, depending on the attribute.

Syntax

object.**Attributes** [= *newattributes*]

The **Attributes** property has these parts:

Part	Description
object	Required. Always the name of a **File** or **Folder** object.
newattributes	Optional. If provided, *newattributes* is the new value for the attributes of the specified *object*.

Settings

The *newattributes* argument can have any of the following values or any logical combination of the following values:

Constant	Value	Description

Normal	0	Normal file. No attributes are set.
ReadOnly	1	Read-only file. Attribute is read/write.
Hidden	2	Hidden file. Attribute is read/write.
System	4	System file. Attribute is read/write.
Volume	8	Disk drive volume label. Attribute is read-only.
Directory	16	Folder or directory. Attribute is read-only.
Archive	32	File has changed since last backup. Attribute is read/write.
Alias	64	Link or shortcut. Attribute is read-only.
Compressed	128	Compressed file. Attribute is read-only.

Remarks

The following code illustrates the use of the **Attributes** property with a file:

```
Sub SetClearArchiveBit(filespec)
    Dim fs, f, r
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(fs.GetFileName(filespec)
    If f.attributes and 32 Then
        r = MsgBox("The Archive bit is set, do
        If r = vbYes Then
            f.attributes = f.attributes - 32
            MsgBox "Archive bit is cleared."
        Else
            MsgBox "Archive bit remains set."
        End If
    Else
        r = MsgBox("The Archive bit is not set.
        If r = vbYes Then
            f.attributes = f.attributes + 32
            MsgBox "Archive bit is set."
```

```
        Else
            MsgBox "Archive bit remains clear."
        End If
    End If
End Sub
```

Description

Returns the amount of space available to a user on the specified drive or network share.

Syntax

object.**AvailableSpace**

The *object* is always a **Drive** object.

Remarks

The value returned by the **AvailableSpace** property is typically the same as that returned by the **FreeSpace** property. Differences may occur between the two values for computer systems that support quotas.

The following code illustrates the use of the **AvailableSpace** property:

```
Sub ShowAvailableSpace(drvPath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set d = fs.GetDrive(fs.GetDriveName(drvPath
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName  & vbCrLf
    s = s & "Available Space: " & FormatNumber(
    s = s & " Kbytes"
    MsgBox s
End Sub
```

Returns or sets a value specifying the type of calendar to use with your project.

You can use one of two settings for **Calendar**:

Setting	Value	Description
vbCalGreg	0	Use Gregorian calendar (default).
vbCalHijri	1	Use Hijri calendar.

Remarks

You can only set the **Calendar** property programmatically. For example, to use the Hijri calendar, use:

```
Calendar = vbCalHijri
```

Description

Read-only property that returns the column number of the current character position in a **TextStream** file.

Syntax

object.**Column**

The *object* is always the name of a **TextStream** object.

Remarks

After a newline character has been written, but before any other character is written, **Column** is equal to 1.

Description

Sets and returns the comparison mode for comparing string keys in a **Dictionary** object.

Syntax

object.**CompareMode**[= *compare*]

The **CompareMode** property has the following parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
compare	Optional. If provided, *compare* is a value representing the comparison mode used by functions such as **StrComp**.

Settings

The *compare* argument can have the following values:

Constant	Value	Description
vbUseCompareOption	–1	Performs a comparison using the setting of

		the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Remarks

An error occurs if you try to change the comparison mode of a **Dictionary** object that already contains data.

The **CompareMode** property uses the same values as the *compare* argument for the **StrComp** function. Values greater than 2 can be used to refer to comparisons using specific Locale IDs (LCID).

Count Property

Returns a Long (long integer) containing the number of objects in a collection. Read-only.

Description

Returns the number of items in a collection or **Dictionary** object. Read-only.

Syntax

object.**Count**

The *object* is always the name of one of the items in the Applies To list.

Remarks

The following code illustrates use of the **Count** property:

```
Dim a, d, i             'Create some variables
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"     'Add some keys and item
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
a = d.Keys              'Get the keys
For i = 0 To d.Count -1 'Iterate the array
    Print a(i)          'Print key
```

Next
...

Description

Returns the date and time that the specified file or folder was created. Read-only.

Syntax

object.**DateCreated**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **DateCreated** property with a file:

```
Sub ShowFileInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = "Created: " & f.DateCreated
    MsgBox s
End Sub
```

Description

Returns the date and time that the specified file or folder was last accessed. Read-only.

Syntax

object.**DateLastAccessed**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **DateLastAccessed** property with a file:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = UCase(filespec) & vbCrLf
    s = s & "Created: " & f.DateCreated & vbCrL
```

```
    s = s & "Last Accessed: " & f.DateLastAcces
    s = s & "Last Modified: " & f.DateLastModif
    MsgBox s, 0, "File Access Info"
End Sub
```

Important This method depends on the underlying operating system for its behavior. If the operating system does not support providing time information, none will be returned.

Description

Returns the date and time that the specified file or folder was last modified. Read-only.

Syntax

object.**DateLastModified**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **DateLastModified** property with a file:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = UCase(filespec) & vbCrLf
    s = s & "Created: " & f.DateCreated & vbCrL
```

```
    s = s & "Last Accessed: " & f.DateLastAcces
    s = s & "Last Modified: " & f.DateLastModif
    MsgBox s, 0, "File Access Info"
End Sub
```

Description Property

Returns or sets a string expression containing a descriptive string associated with an object. Read/write.

For the **Err** object, returns or sets a descriptive string associated with an error.

Remarks

The **Description** property setting consists of a short description of the error. Use this property to alert the user to an error that you either can't or don't want to

handle. When generating a user-defined error, assign a short description of your error to the **Description** property. If **Description** isn't filled in, and the value of **Number** corresponds to a Visual Basic run-time error, the string returned by the **Error** function is placed in **Description** when the error is generated.

Description

Returns the drive letter of the drive on which the specified file or folder resides. Read-only.

Syntax

object.**Drive**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **Drive** property:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = f.Name & " on Drive " & UCase(f.Drive) 
    s = s & "Created: " & f.DateCreated & vbCrL
    s = s & "Last Accessed: " & f.DateLastAcces
```

```
    s = s & "Last Modified: " & f.DateLastModif
    MsgBox s, 0, "File Access Info"
End Sub
```

Description

Returns the drive letter of a physical local drive or a network share. Read-only.

Syntax

object.**DriveLetter**

The *object* is always a **Drive** object.

Remarks

The **DriveLetter** property returns a zero-length string ("") if the specified drive is not associated with a drive letter, for example, a network share that has not been mapped to a drive letter.

The following code illustrates the use of the **DriveLetter** property:

```
Sub ShowDriveLetter(drvPath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(fs.GetDriveName(drvPath
```

```
    s = "Drive " & d.DriveLetter & ": - "
    s = s & d.VolumeName  & vbCrLf
    s = s & "Free Space: " & FormatNumber(d.Fre
    s = s & " Kbytes"
    MsgBox s
End Sub
```

Description

Returns a **Drives** collection consisting of all **Drive** objects available on the local machine.

Syntax

object.**Drives**

The *object* is always a **FileSystemObject**.

Remarks

Removable-media drives need not have media inserted for them to appear in the **Drives** collection.

You can iterate the members of the **Drives** collection using a **For Each...Next** construct as illustrated in the following code:

```
Sub ShowDriveList
    Dim fs, d, dc, s, n
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set dc = fs.Drives
    For Each d in dc
        s = s & d.DriveLetter & " - "
        If d.DriveType = 3 Then
            n = d.ShareName
        Else
            n = d.VolumeName
        End If
        s = s & n & vbCrLf
    Next
    MsgBox s
End Sub
```

Description

Returns a value indicating the type of a specified drive.

Syntax

object.**DriveType**

The *object* is always a **Drive** object.

Remarks

The following code illustrates the use of the **DriveType** property:

```
Sub ShowDriveType(drvpath)
    Dim fs, d, s, t
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(drvpath)
    Select Case d.DriveType
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
```

```
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
    s = "Drive " & d.DriveLetter & ": - " & t
    MsgBox s
End Sub
```

Description

Returns a **Files** collection consisting of all **File** objects contained in the specified folder, including those with hidden and system file attributes set.

Syntax

object.**Files**

The *object* is always a **Folder** object.

Remarks

The following code illustrates the use of the **Files** property:

```
Sub ShowFileList(folderspec)
    Dim fs, f, f1, fc, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(folderspec)
    Set fc = f.Files
    For Each f1 in fc
        s = s & f1.name
```

```
        s = s &  vbCrLf
    Next
    MsgBox s
End Sub
```

Description

Returns the type of file system in use for the specified drive.

Syntax

object.**FileSystem**

The *object* is always a **Drive** object.

Remarks

Available return types include FAT, NTFS, and CDFS.

The following code illustrates the use of the **FileSystem** property:

```
Sub ShowFileSystemType
    Dim fs,d, s
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive("e:")
    s = d.FileSystem
    MsgBox s
```

```
End Sub
```

Description

Returns the amount of free space available to a user on the specified drive or network share. Read-only.

Syntax

object.**FreeSpace**

The *object* is always a **Drive** object.

Remarks

The value returned by the **FreeSpace** property is typically the same as that returned by the **AvailableSpace** property. Differences may occur between the two for computer systems that support quotas.

The following code illustrates the use of the **FreeSpace** property:

```
Sub ShowFreeSpace(drvPath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set d = fs.GetDrive(fs.GetDriveName(drvPath
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName  & vbCrLf
    s = s & "Free Space: " & FormatNumber(d.Fre
    s = s & " Kbytes"
    MsgBox s
End Sub
```

Returns or sets a string expression containing the context ID for a topic in a Help file. Read/write.

Remarks

The **HelpContext** property is used to automatically display the Help topic specified in the **HelpFile** property. If both **HelpFile** and **HelpContext** are empty, the value of **Number** is checked. If **Number** corresponds to a Visual Basic run-time error value, then the Visual Basic Help context ID for the error is used. If

the **Number** value doesn’t correspond to a Visual Basic error, the contents screen for the Visual Basic Help file is displayed.

Note You should write routines in your application to handle typical errors. When programming with an object, you can use the object's Help file to improve the quality of your error handling, or to display a meaningful message to your user if the error isn’t recoverable.

HelpFile Property

Returns or sets a string expression the fully qualified path to a Help file. Read/write.

Remarks

If a Help file is specified in **HelpFile**, it is automatically called when the user presses the **Help** button (or the F1 KEY in Windows or the HELP key on the Macintosh) in the error message dialog box. If the **HelpContext** property contains a valid context ID for the specified file, that topic is automatically displayed. If no **HelpFile** is specified, the Visual Basic Help file is displayed.

Note You should write routines in your application to handle typical errors. When programming with an object, you can use the object's Help file to improve the quality of your error handling, or to display a meaningful message to your user if the error isn’t recoverable.

Description

Returns **True** if the specified drive is ready; **False** if it is not.

Syntax

object.**IsReady**

The object is always a **Drive** object.

Remarks

For removable-media drives and CD-ROM drives, **IsReady** returns **True** only when the appropriate media is inserted and ready for access.

The following code illustrates the use of the **IsReady** property:

```
Sub ShowDriveInfo(drvpath)
    Dim fs, d, s, t
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(drvpath)
    Select Case d.DriveType
```

```
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
    s = "Drive " & d.DriveLetter & ": - " & t
    If d.IsReady Then
        s = s & vbCrLf & "Drive is Ready."
    Else
        s = s & vbCrLf & "Drive is not Ready."
    End If
    MsgBox s
End Sub
```

Description

Returns **True** if the specified folder is the root folder; **False** if it is not.

Syntax

object.**IsRootFolder**

The *object* is always a **Folder** object.

Remarks

The following code illustrates the use of the **IsRootFolder** property:

```
Dim fs
Set fs = CreateObject("Scripting.FileSystemObje
Sub DisplayLevelDepth(pathspec)
    Dim f, n
    Set f = fs.GetFolder(pathspec)
    If f.IsRootFolder Then
        MsgBox "The specified folder is the roo
    Else
```

```
        Do Until f.IsRootFolder
            Set f = f.ParentFolder
            n = n + 1
        Loop
        MsgBox "The specified folder is nested
    End If
End Sub
```

Description

Sets or returns an *item* for a specified *key* in a **Dictionary** object. For collections, returns an *item* based on the specified *key*. Read/write.

Syntax

object.**Item(***key***)** [= *newitem*]

The **Item** property has the following parts:

Part	Description
object	Required. Always the name of a collection or **Dictionary** object.
key	Required. *Key* associated with the item being retrieved or added.
newitem	Optional. Used for **Dictionary** object only; no application for collections. If provided, *newitem* is the new value associated with the specified *key*.

Remarks

If *key* is not found when changing an *item*, a new *key* is created with the specified *newitem*. If *key* is not found when attempting to return an existing item,

a new *key* is created and its corresponding item is left empty.

Description

Sets a *key* in a **Dictionary** object.

Syntax

object.**Key(***key***)** = *newkey*

The **Key** property has the following parts:

Part	Description
object	Required. Always the name of a **Dictionary** object.
key	Required. *Key* value being changed.
newkey	Required. New value that replaces the specified *key*.

Remarks

If *key* is not found when changing a *key*, a run-time error will occur.

Returns a system error code produced by a call to a dynamic-link library (DLL). Read-only. LastDLLError always returns zero on the Macintosh.

Remarks

The **LastDLLError** property applies only to DLL calls made from Visual Basic code. When such a call is made, the called function usually returns a code indicating success or failure, and the **LastDLLError** property is filled. Check the documentation for the DLL's functions to determine the return values that indicate success or failure. Whenever the failure code is returned, the Visual Basic application should immediately check the **LastDLLError** property. No exception is raised when the **LastDLLError** property is set.

Description

Read-only property that returns the current line number in a **TextStream** file.

Syntax

object.**Line**

The *object* is always the name of a **TextStream** object.

Remarks

After a file is initially opened and before anything is written, **Line** is equal to 1.

Description

Sets or returns the name of a specified file or folder. Read/write.

Syntax

object.**Name** [= *newname*]

The **Name** property has these parts:

Part	Description
object	Required. Always the name of a **File** or **Folder** object.
newname	Optional. If provided, *newname* is the new name of the specified *object*.

Remarks

The following code illustrates the use of the **Name** property:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set f = fs.GetFile(filespec)
    s = f.Name & " on Drive " & UCase(f.Drive) 
    s = s & "Created: " & f.DateCreated & vbCrL
    s = s & "Last Accessed: " & f.DateLastAcces
    s = s & "Last Modified: " & f.DateLastModif
    MsgBox s, 0, "File Access Info"
End Sub
```

Number Property

Returns or sets a numeric value specifying an error. **Number** is the **Err** object's default property. Read/write.

Remarks

When returning a user-defined error from an object, set **Err.Number** by adding the number you selected as an error code to the **vbObjectError** constant. For example, you use the following code to return the number 1051 as an error code:

```
Err.Raise Number := vbObjectError + 1051, Sourc
```

Description

Returns the folder object for the parent of the specified file or folder. Read-only.

Syntax

object.**ParentFolder**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **ParentFolder** property with a file:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = UCase(f.Name) & " in " & UCase(f.Parent
    s = s & "Created: " & f.DateCreated & vbCrL
    s = s & "Last Accessed: " & f.DateLastAcces
    s = s & "Last Modified: " & f.DateLastModif
```

```
    MsgBox s, 0, "File Access Info"
End Sub
```

Description

Returns the path for a specified file, folder, or drive.

Syntax

object.**Path**

The *object* is always a **File**, **Folder**, or **Drive** object.

Remarks

For drive letters, the root drive is not included. For example, the path for the C drive is C:, not C:\.

The following code illustrates the use of the **Path** property with a **File** object:

```
Sub ShowFileAccessInfo(filespec)
    Dim fs, d, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = UCase(f.Path) & vbCrLf
```

```
    s = s & "Created: " & f.DateCreated & vbCrL
    s = s & "Last Accessed: " & f.DateLastAcces
    s = s & "Last Modified: " & f.DateLastModif
    MsgBox s, 0, "File Access Info"
End Sub
```

Description

Returns a **Folder** object representing the root folder of a specified drive. Read-only.

Syntax

object.**RootFolder**

The *object* is always a **Drive** object.

Remarks

All the files and folders contained on the drive can be accessed using the returned **Folder** object.

Description

Returns the decimal serial number used to uniquely identify a disk volume.

Syntax

object.**SerialNumber**

The *object* is always a **Drive** object.

Remarks

You can use the **SerialNumber** property to ensure that the correct disk is inserted in a drive with removable media.

The following code illustrates the use of the **SerialNumber** property:

```
Sub ShowDriveInfo(drvpath)
    Dim fs, d, s, t
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(fs.GetDriveName(fs.GetA
    Select Case d.DriveType
```

```
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
    s = "Drive " & d.DriveLetter & ": - " & t
    s = s & vbCrLf & "SN: " & d.SerialNumber
    MsgBox s
End Sub
```

Description

Returns the network share name for a specified drive.

Syntax

object.**ShareName**

The *object* is always a **Drive** object.

Remarks

If *object* is not a network drive, the **ShareName** property returns a zero-length string ("").

The following code illustrates the use of the **ShareName** property:

```
Sub ShowDriveInfo(drvpath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(fs.GetDriveName(fs.GetA
    s = "Drive " & d.DriveLetter & ": - " & d.S
```

```
    MsgBox s
End Sub
```

Description

Returns the short name used by programs that require the earlier 8.3 naming convention.

Syntax

object.**ShortName**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **ShortName** property with a **File** object:

```
Sub ShowShortName(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = "The short name for " & "" & UCase(f.Na
    s = s & "" & vbCrLf
```

```
    s = s & "is: " & "" & f.ShortName & ""
    MsgBox s, 0, "Short Name Info"
End Sub
```

Description

Returns the short path used by programs that require the earlier 8.3 file naming convention.

Syntax

object.**ShortPath**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **ShortName** property with a **File** object:

```
Sub ShowShortPath(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFile(filespec)
    s = "The short path for " & "" & UCase(f.Na
    s = s & "" & vbCrLf
```

```
    s = s & "is: " & "" & f.ShortPath & ""
    MsgBox s, 0, "Short Path Info"
End Sub
```

Description

For files, returns the size, in bytes, of the specified file. For folders, returns the size, in bytes, of all files and subfolders contained in the folder.

Syntax

object.**Size**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **Size** property with a **Folder** object:

```
Sub ShowFolderSize(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(filespec)
    s = UCase(f.Name) & " uses " & f.size & " b
    MsgBox s, 0, "Folder Size Info"
End Sub
```

Source Property

Returns or sets a string expression specifying the name of the object or application that originally generated the error. Read/write.

Remarks

The **Source** property specifies a string expression representing the object that generated the error; the expression is usually the object's class name or programmatic ID. Use **Source** to provide information when your code is unable to handle an error generated in an accessed object. For example, if you access Microsoft Excel and it generates a `Division by zero` error, Microsoft Excel sets **Err.Number** to its error code for that error and sets **Source** to Excel.Application.

When generating an error from code, **Source** is your application's programmatic ID. For class modules, **Source** should contain a name having the form *project.class*. When an unexpected error occurs in your code, the **Source** property is automatically filled in. For errors in a standard module, **Source** contains the project name. For errors in a class module, **Source** contains a name with the *project.class* form.

Returns or sets a value specifying the position of a **UserForm** when it first appears.

You can use one of four settings for **StartUpPosition**:

Setting	Value	Description
Manual	0	No initial setting specified.
CenterOwner	1	Center on the item to which the **UserForm** belongs.
CenterScreen	2	Center on the whole screen.
WindowsDefault	3	Position in upper-left corner of screen.

Remarks

You can set the **StartUpPosition** property programmatically or from the **Properties** window.

Description

Returns a **Folders** collection consisting of all folders contained in a specified folder, including those with Hidden and System file attributes set.

Syntax

object.**SubFolders**

The *object* is always a **Folder** object.

Remarks

The following code illustrates the use of the **SubFolders** property:

```
Sub ShowFolderList(folderspec)
    Dim fs, f, f1, s, sf
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(folderspec)
    Set sf = f.SubFolders
    For Each f1 in sf
        s = s & f1.name
```

```
        s = s &  vbCrLf
    Next
    MsgBox s
End Sub
```

Description

Returns the total space, in bytes, of a drive or network share.

Syntax

object.**TotalSize**

The *object* is always a **Drive** object.

Remarks

The following code illustrates the use of the **TotalSize** property:

```
Sub ShowSpaceInfo(drvpath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
    Set d = fs.GetDrive(fs.GetDriveName(fs.GetA
    s = "Drive " & d.DriveLetter & ":"
    s = s & vbCrLf
    s = s & "Total Size: " & FormatNumber(d.Tot
    s = s & vbCrLf
```

```
    s = s & "Available: " & FormatNumber(d.Avai
    MsgBox s
End Sub
```

Description

Returns information about the type of a file or folder. For example, for files ending in .TXT, "Text Document" is returned.

Syntax

object.**Type**

The *object* is always a **File** or **Folder** object.

Remarks

The following code illustrates the use of the **Type** property to return a folder type. In this example, try providing the path of the Recycle Bin or other unique folder to the procedure.

```
Sub ShowFileSize(filespec)
    Dim fs, f, s
    Set fs = CreateObject("Scripting.FileSystem
    Set f = fs.GetFolder(filespec)
    s = UCase(f.Name) & " is a " & f.Type
```

```
    MsgBox s, 0, "File Size Info"
End Sub
```

Description

Sets or returns the volume name of the specified drive. Read/write.

Syntax

object.**VolumeName** [= *newname*]

The VolumeName property has these parts:

Part	Description
object	Required. Always the name of a **Drive** object.
newname	Optional. If provided, *newname* is the new name of the specified *object*.

Remarks

The following code illustrates the use of the **VolumeName** property:

```
Sub ShowVolumeInfo(drvpath)
    Dim fs, d, s
    Set fs = CreateObject("Scripting.FileSystem
```

```
    Set d = fs.GetDrive(fs.GetDriveName(fs.GetA
    s = "Drive " & d.DriveLetter & ": - " & d.V
    MsgBox s
End Sub
```

AppActivate Statement

Activates an application window.

Syntax

AppActivate ***title***[, ***wait***]

The **AppActivate** statement syntax has these named arguments:

Part	Description

title Required. String expression specifying the title in the title bar of the application window you want to activate. The task ID returned by the **Shell** function can be used in place of ***title*** to activate an application.

wait Optional. Boolean value specifying whether the calling application has the focus before activating another. If **False** (default), the specified application is immediately activated, even if the calling application does not have the focus. If **True**, the calling application waits until it has the focus, then activates the specified application.

Remarks

The **AppActivate** statement changes the focus to the named application or window but does not affect whether it is maximized or minimized. Focus moves from the activated application window when the user takes some action to change the focus or close the window. Use the **Shell** function to start an application and set the window style.

In determining which application to activate, ***title*** is compared to the title string of each running application. If there is no exact match, any application whose title string begins with ***title*** is activated. If there is more than one instance of the application named by ***title***, one instance is arbitrarily activated.

Sounds a tone through the computer's speaker.

Syntax

Beep

Remarks

The frequency and duration of the beep depend on your hardware and system software, and vary among computers.

Call Statement

Transfers control to a **Sub** procedure, **Function** procedure, or dynamic-link library (DLL) procedure.

Syntax

[**Call**] *name* [*argumentlist*]

The **Call** statement syntax has these parts:

Part	Description
Call	Optional; keyword. If specified, you must enclose *argumentlist* in parentheses. For example: `Call MyProc(0)`
name	Required. Name of the procedure to call.
argumentlist	Optional. Comma-delimited list of variables, arrays, or expressions to pass to the procedure. Components of *argumentlist* may include the keywords **ByVal** or **ByRef** to describe how the arguments are treated by the called procedure. However, **ByVal** and **ByRef** can be used with **Call** only when calling a DLL procedure. On the Macintosh, **ByVal** and **ByRef** can be used with **Call** when making a call to a Macintosh code resource.

Remarks

You are not required to use the **Call** keyword when calling a procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, *argumentlist* must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around *argumentlist*. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's

return value is discarded.

To pass a whole array to a procedure, use the array name followed by empty parentheses.

ChDir Statement

Changes the current directory or folder.

Syntax

ChDir *path*

The required *path* argument is a string expression that identifies which directory or folder becomes the new default directory or folder. The *path* may include the drive. If no drive is specified, **ChDir** changes the default directory or folder on the current drive.

Remarks

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"
```

On the Power Macintosh, the default drive always changes to the drive specified in *path*. Full path specifications begin with the volume name, and relative paths begin with a colon (**:**). **ChDir** resolves any aliases specified in the path:

```
ChDir "MacDrive:Tmp"    ' On the Macintosh.
```

Note that when making relative directory changes, different symbols are used in Microsoft Windows and on the Macintosh:

```
ChDir ".."    ' Moves up one directory in Micro
ChDir "::"    ' Moves up one directory on the M
```

Changes the current drive.

Syntax

ChDrive *drive*

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

On the Macintosh, **ChDrive** changes the current folder to the root folder of the specified drive.

Close Statement

Concludes input/output (I/O) to a file opened using the **Open** statement.

Syntax

Close [*filenumberlist*]

The optional *filenumberlist* argument can be one or more file numbers using the following syntax, where *filenumber* is any valid file number:

[[#]*filenumber*] [**,** [#]*filenumber*] **. . .**

Remarks

If you omit *filenumberlist*, all active files opened by the **Open** statement are closed.

When you close files that were opened for **Output** or **Append**, the final buffer of output is written to the operating system buffer for that file. All buffer space associated with the closed file is released.

When the **Close** statement is executed, the association of a file with its file number ends.

Const Statement

Declares constants for use in place of literal values.

Syntax

[**Public** | **Private**] **Const** *constname* [**As** *type*] = *expression*

The **Const** statement syntax has these parts:

Part	Description
Public	Optional. Keyword used at module level to declare constants that are available to all procedures in all modules. Not allowed in procedures.
Private	Optional. Keyword used at module level to declare constants that are available only within the module where the declaration is made. Not allowed in procedures.
constname	Required. Name of the constant; follows standard variable

	naming conventions.
type	Optional. Data type of the constant; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or Variant. Use a separate **As** *type* clause for each constant being declared.
expression	Required. Literal, other constant, or any combination that includes all arithmetic or logical operators except **Is**.

Remarks

Constants are private by default. Within procedures, constants are always private; their visibility can't be changed. In standard modules, the default visibility of module-level constants can be changed using the **Public** keyword. In class modules, however, constants can only be private and their visibility can't be changed using the **Public** keyword.

To combine several constant declarations on the same line, separate each constant assignment with a comma. When constant declarations are combined in this way, the **Public** or **Private** keyword, if used, applies to all of them.

You can't use variables, user-defined functions, or intrinsic Visual Basic functions (such as **Chr**) in expressions assigned to constants.

Note Constants can make your programs self-documenting and easy to modify. Unlike variables, constants can't be inadvertently changed while your program is running.

If you don't explicitly declare the constant type using **As** *type*, the constant has the data type that is most appropriate for *expression*.

Constants declared in a **Sub**, **Function**, or **Property** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the module in which it is declared. You can use constants anywhere you can use an expression.

Sets the current system date.

Syntax

Date = *date*

For systems running Microsoft Windows 95, the required *date* specification must be a date from January 1, 1980 through December 31, 2099. For systems running Microsoft Windows NT, *date* must be a date from January 1, 1980 through December 31, 2079. For the Macintosh, *date* must be a date from January 1, 1904 through February 5, 2040.

Declare Statement

Used at module level to declare references to external procedures in a dynamic-link library (DLL).

Syntax 1

[**Public** | **Private**] **Declare Sub** *name* **Lib** "*libname*" [**Alias** "*aliasname*"] [**(**[*arglist*]**)**]

Syntax 2

[**Public** | **Private**] **Declare Function** *name* **Lib** "*libname*" [**Alias** "*aliasname*"] [**(**[*arglist*]**)**] [**As** *type*]

The **Declare** statement syntax has these parts:

Part	Description
Public	Optional. Used to declare procedures that are available to all other procedures in all modules.
Private	Optional. Used to declare procedures that are available only within the module where the declaration is made.
Sub	Optional (either **Sub** or **Function** must appear). Indicates that the procedure doesn't return a value.
Function	Optional (either **Sub** or **Function** must appear). Indicates that the procedure returns a value that can be used in an expression.
name	Required. Any valid procedure name. Note that DLL entry points are case sensitive.
Lib	Required. Indicates that a DLL or code resource contains the procedure being declared. The **Lib** clause is required for all declarations.
libname	Required. Name of the DLL or code resource that contains the

	declared procedure.
Alias	Optional. Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as a keyword. You can also use **Alias** when a DLL procedure has the same name as a public variable, constant, or any other procedure in the same scope. **Alias** is also useful if any characters in the DLL procedure name aren't allowed by the DLL naming convention.
aliasname	Optional. Name of the procedure in the DLL or code resource. If the first character is not a number sign (#), *aliasname* is the name of the procedure's entry point in the DLL. If (#) is the first character, all characters that follow must indicate the ordinal number of the procedure's entry point.
arglist	Optional. List of variables representing arguments that are passed to the procedure when it is called.
type	Optional. Data type of the value returned by a **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), or Variant, a user-defined type, or an object type.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an

	arbitrary number of arguments. The **ParamArray** keyword can't be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.
( )	Required for array variables. Indicates that *varname* is an array.
type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported), **Date**, **String** (variable length only), **Object**, **Variant**, a user-defined type, or an object type.

Remarks

For **Function** procedures, the data type of the procedure determines the data type it returns. You can use an **As** clause following *arglist* to specify the return type of the function. Within *arglist*, you can use an **As** clause to specify the data type of any of the arguments passed to the procedure. In addition to specifying any of the standard data types, you can specify **As Any** in *arglist* to inhibit type checking and allow any data type to be passed to the procedure.

Empty parentheses indicate that the **Sub** or **Function** procedure has no arguments and that Visual Basic should 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" ()
```

If you include an argument list, 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 As Long)
```

Note You can't have fixed-length strings in the argument list of a **Declare** statement; 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.

Note The **vbNullString** constant is used when calling external procedures, where the external procedure requires a string whose value is zero. This is not the same thing as a zero-length string ("").

Deftype Statements

Used at module level to set the default data type for variables, arguments passed to procedures, and the return type for **Function** and **Property Get** procedures whose names start with the specified characters.

Syntax

DefBool *letterrange*[**,** *letterrange*] **. . .**

DefByte *letterrange*[**,** *letterrange*] **. . .**

DefInt *letterrange*[**,** *letterrange*] **. . .**

DefLng *letterrange*[**,** *letterrange*] **. . .**

DefCur *letterrange*[**,** *letterrange*] **. . .**

DefSng *letterrange*[**,** *letterrange*] **. . .**

DefDbl *letterrange*[**,** *letterrange*] **. . .**

DefDec *letterrange*[**,** *letterrange*] **. . .**

DefDate *letterrange*[**,** *letterrange*] **. . .**

DefStr *letterrange*[**,** *letterrange*] **. . .**

DefObj *letterrange*[**,** *letterrange*] **. . .**

DefVar *letterrange*[**,** *letterrange*] **. . .**

The required *letterrange* argument has the following syntax:

letter1[**-***letter2*]

The *letter1* and *letter2* arguments specify the name range for which you can set a default data type. Each argument represents the first letter of the variable, argument, **Function** procedure, or **Property Get** procedure name and can be any letter of the alphabet. The case of letters in *letterrange* isn't significant.

Remarks

The statement name determines the data type:

Statement	Data Type
DefBool	Boolean
DefByte	Byte
DefInt	Integer
DefLng	Long
DefCur	Currency
DefSng	Single
DefDbl	Double
DefDec	Decimal (not currently supported)
DefDate	Date
DefStr	String
DefObj	Object
DefVar	Variant

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 where it is used. For example, a **DefInt** statement in one module affects only the default data type of variables, arguments passed to procedures, and the return type for **Function** and **Property Get** procedures declared in that module; the default data type of variables, arguments, and return types in other modules is unaffected. If not explicitly declared with a **Def***type* statement, the default data type for all variables, all arguments, all **Function** procedures, and all **Property Get** procedures is **Variant**.

When you specify a letter range, it usually defines the data type for variables that begin with letters in the first 128 characters of the character set. However, when you specify the letter range A–Z, you set the default to the specified data type for all variables, including variables that begin with international characters from the extended part of the character set (128–255).

Once the range A–Z has been specified, you can't further redefine any subranges of variables using **Def***type* statements. Once a range has been specified, if you include a previously defined letter in another **Def***type* statement, an error occurs. However, you can explicitly specify the data type of any variable, defined or not, using a **Dim** statement with an **As** *type* clause. For example, you can use the following code at module level to define a variable as a **Double** even though the default data type is **Integer**:

```
DefInt A-Z
Dim TaxRate As Double
```

Def*type* statements don't affect elements of user-defined types because the elements must be explicitly declared.

DeleteSetting Statement

Deletes a section or key setting from an application's entry in the Windows registry or (on the Macintosh) information in the application’s initialization file.

Syntax

DeleteSetting ***appname*****,** ***section***[**,** ***key***]

The **DeleteSetting** statement syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project to which the section or key setting applies. On the Macintosh, this is the filename of the initialization file in the Preferences folder in the System folder.
section	Required. String expression containing the name of the section where the key setting is being deleted. If only ***appname*** and ***section*** are provided, the specified section is deleted along with all related key settings.
key	Optional. String expression containing the name of the key setting being deleted.

Remarks

If all arguments are provided, the specified setting is deleted. A run-time error occurs if you attempt to use the **DeleteSetting** statement on a non-existent section or key setting.

Dim Statement

Declares variables and allocates storage space.

Syntax

Dim [**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*] [**,** [**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*]] **. . .**

The **Dim** statement syntax has these parts:

Part	Description
WithEvents	Optional. Keyword that specifies that *varname* is an object variable used to respond to events triggered by an ActiveX object. **WithEvents** is valid only in class modules. You can declare as many individual variables as you like using **WithEvents**, but you can't create arrays with **WithEvents**. You can't use **New** with **WithEvents**.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,** [*lower* **To**] *upper*] **. . .** When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type, can't be used to declare instances of dependent objects, and can’t be used with **WithEvents**.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable you declare.

Remarks

Variables declared with **Dim** at the module level are available to all procedures within the module. At the procedure level, variables are available only within the procedure.

Use the **Dim** statement at module or procedure level to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**.

```
Dim NumberOfEmployees As Integer
```

Also use a **Dim** statement to declare the object type of a variable. The following declares a variable for a new instance of a worksheet.

```
Dim X As New Worksheet
```

If the **New** keyword is not used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

You can also use the **Dim** statement with empty parentheses to declare a dynamic array. 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 was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.

If you don't specify a data type or object type, and there is no **Def***type* statement in the module, the variable is **Variant** by default.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Note When you use the **Dim** statement in a procedure, you generally put the **Dim** statement at the beginning of the procedure.

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax

Do [{**While** | **Until**} *condition*]
  [*statements*]
  [**Exit Do**]
  [*statements*]

Loop

Or, you can use this syntax:

Do
[*statements*]
[**Exit Do**]
[*statements*]

Loop [{**While** | **Until**} *condition*]

The **Do Loop** statement syntax has these parts:

Part	Description
condition	Optional. Numeric expression or string expression that is **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	One or more statements that are repeated while, or until, *condition* is **True**.

Remarks

Any number of **Exit Do** statements may be placed anywhere in the **Do…Loop** as an alternate way to exit a **Do…Loop**. **Exit Do** is often used after evaluating some condition, for example, **If…Then**, in which case the **Exit Do** statement transfers control to the statement immediately following the **Loop**.

When used within nested **Do…Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.

End Statement

Ends a procedure or block.

Syntax

End

End Function

End If

End Property

End Select

End Sub

End Type

End With

The **End** statement syntax has these forms:

Statement	Description
End	Terminates execution immediately. Never required by itself but may be placed anywhere in a procedure to end code execution, close files opened with the **Open** statement and to clear variables.
End Function	Required to end a **Function** statement.
End If	Required to end a block **If…Then…Else** statement.
End Property	Required to end a **Property Let**, **Property Get**, or **Property Set** procedure.

End Select	Required to end a **Select Case** statement.
End Sub	Required to end a **Sub** statement.
End Type	Required to end a user-defined type definition (**Type** statement).
End With	Required to end a **With** statement.

Remarks

When executed, the **End** statement resets all module-level variables and all static local variables in all modules. To preserve the value of these variables, use the **Stop** statement instead. You can then resume execution while preserving the value of those variables.

Note The **End** statement stops code execution abruptly, without invoking the Unload, QueryUnload, or Terminate event, or any other Visual Basic code. Code you have placed in the Unload, QueryUnload, and Terminate events of forms and class modules is not executed. Objects created from class modules are destroyed, files opened using the **Open** statement are closed, and memory used by your program is freed. Object references held by other programs are invalidated.

The **End** statement provides a way to force your program to halt. For normal termination of a Visual Basic program, you should unload all forms. Your program closes as soon as there are no other programs holding references to objects created from your public class modules and no code executing.

Erase Statement

Reinitializes the elements of fixed-size arrays and releases dynamic-array storage space.

Syntax

Erase *arraylist*

The required *arraylist* argument is one or more comma-delimited array variables to be erased.

Remarks

Erase behaves differently depending on whether an array is fixed-size (ordinary) or dynamic. **Erase** recovers no memory for fixed-size arrays. **Erase** sets the elements of a fixed array as follows:

Type of Array	Effect of Erase on Fixed-Array Elements
Fixed numeric array	Sets each element to zero.
Fixed string array (variable length)	Sets each element to a zero-length string ("").
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.
Array of objects	Sets each element to the special value **Nothing**.

Erase frees the memory used by dynamic arrays. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a **ReDim** statement.

Error Statement

Simulates the occurrence of an error.

Syntax

Error *errornumber*

The required *errornumber* can be any valid error number.

Remarks

The **Error** statement is supported for backward compatibility. In new code, especially when creating objects, use the **Err** object's **Raise** method to generate run-time errors.

If *errornumber* is defined, the **Error** statement calls the error handler after the properties of **Err** object are assigned the following default values:

Property	Value
Number	Value specified as argument to **Error** statement. Can be any valid error number.
Source	Name of the current Visual Basic project.
Description	String expression corresponding to the return value of the **Error** function for the specified **Number**, if this string exists. If the string doesn't exist, **Description** contains a zero-length string ("").
HelpFile	The fully qualified drive, path, and file name of the appropriate Visual Basic Help file.
HelpContext	The appropriate Visual Basic Help file context ID for the error corresponding to the **Number** property.
LastDLLError	Zero.

If no error handler exists or if none is enabled, an error message is created and displayed from the **Err** object properties.

Note Not all Visual Basic host applications can create objects. See your host application's documentation to determine whether it can create classes and objects.

Exit

Statement

Exits a block of **Do…Loop**, **For...Next**, **Function**, **Sub**, or **Property** code.

Syntax

Exit Do

Exit For

Exit Function

Exit Property

Exit Sub

The **Exit** statement syntax has these forms:

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 where **Exit Do** occurs.
Exit For	Provides a way to exit a **For** loop. It can be used only in a **For...Next** or **For Each...Next** loop. **Exit For** transfers control to the statement following the **Next** statement. When used within nested **For** loops, **Exit For** transfers control to the loop that is one nested level above the loop where **Exit For** occurs.
Exit Function	Immediately exits the **Function** procedure in which it appears. Execution continues with the statement following the statement that called the **Function**.
Exit Property	Immediately exits the **Property** procedure in which it appears. Execution continues with the statement following the statement that called the **Property** procedure.
Exit Sub	Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub** procedure.

Remarks

Do not confuse **Exit** statements with **End** statements. **Exit** does not define the end of a structure.

FileCopy Statement

Copies a file.

Syntax

FileCopy *source, destination*

The **FileCopy** statement syntax has these named arguments:

Part	Description
source	Required. String expression that specifies the name of the file to be copied. The ***source*** may include directory or folder, and drive.
destination	Required. String expression that specifies the target file name. The ***destination*** may include directory or folder, and drive.

Remarks

If you try to use the **FileCopy** statement on a currently open file, an error occurs.

For Each...Next Statement

Repeats a group of statements for each element in an array or collection.

Syntax

For Each *element* **In** *group* [*statements*]
  [**Exit For**]
  [*statements*]

Next [*element*]

The **For...Each...Next** statement syntax has these parts:

Part	Description
element	Required. Variable used to iterate through the elements of the collection or array. For collections, *element* can only be a Variant variable, a generic object variable, or any specific object variable. For arrays, *element* can only be a **Variant** variable.
group	Required. Name of an object collection or array (except an array of user-defined types).
statements	Optional. One or more statements that are executed on each item in *group*.

Remarks

The **For...Each** block is entered if there is at least one element in *group*. Once the loop has been entered, all the statements in the loop are executed for the first element in *group*. If there are more elements in *group*, the statements in the loop continue to execute for each element. When there are no more elements in *group*, the loop is exited and execution continues with the statement following the **Next** statement.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternative way to exit. **Exit For** is often used after evaluating some condition, for example **If…Then**, and transfers control to the statement immediately following **Next**.

You can nest **For...Each...Next** loops by placing one **For...Each...Next** loop within another. However, each loop *element* must be unique.

Note If you omit *element* in a **Next** statement, execution continues as if *element* is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

You can't use the **For...Each...Next** statement with an array of user-defined types because a **Variant** can't contain a user-defined type.

For...Next Statement

Repeats a group of statements a specified number of times.

Syntax

For *counter* = *start* **To** *end* [**Step** *step*]
  [*statements*]
  [**Exit For**]
  [*statements*]

Next [*counter*]

The **For…Next** statement syntax has these parts:

Part	Description
counter	Required. Numeric variable used as a loop counter. The variable can't be a Boolean or an array element.
start	Required. Initial value of *counter*.
end	Required. Final value of *counter*.
step	Optional. Amount *counter* is changed each time through the loop. If not specified, *step* defaults to one.
statements	Optional. One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The *step* argument can be either positive or negative. The value of the *step* argument determines loop processing as follows:

Value	Loop executes if
Positive or 0	*counter* <= *end*
Negative	*counter* >= *end*

After all 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 initially), 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 it more

difficult to read and debug your code.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternate way to exit. **Exit For** is often used after evaluating of some condition, for example **If...Then**, and transfers control to the statement immediately following **Next**.

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
```

Note If you omit *counter* in a **Next** statement, execution continues as if *counter* is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

Function Statement

Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Function** *name* [**(***arglist***)**] [**As** *type*]
  [*statements*]
  [*name* = *expression*]
  [**Exit Function**]
  [*statements*]
  [*name* = *expression*]

End Function

The **Function** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Function** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private**, the procedure is not available outside the project.
Private	Optional. Indicates that the **Function** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Function** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. 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**, even if they are used in the procedure.

name	Required. Name of the **Function**; follows standard variable naming conventions.
arglist	Optional. List of variables representing arguments that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas.
type	Optional. Data type of the value returned by the **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or (except fixed length), Object, Variant, or any user-defined type.
statements	Optional. Any group of statements to be executed within the **Function** procedure.
expression	Optional. Return value of the **Function**.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.

type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported) **Date**, **String** (variable length only), **Object**, **Variant**, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Function** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure does't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

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** keyword usually isn't used with recursive **Function** procedures.

All executable code must be in procedures. You can't define a **Function** procedure inside another **Function**, **Sub**, or **Property** procedure.

The **Exit Function** statement 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.

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, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function,

such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the **Call** statement for specific information on how to call **Function** procedures.

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *name*, 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. A function that returns an object reference returns **Nothing** if no object reference is assigned to *name* (using **Set**) within the **Function**.

The following example shows how to assign a return value to a function named `BinarySearch`. In this case, **False** is assigned to the name to indicate that some value was not found.

```
Function BinarySearch(. . .) As Boolean
. . .
    ' 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. Variables that are 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 naming conflict can occur if anything you defined at the module level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant, or variable, it is assumed that your procedure refers 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.

Caution Visual 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.

Get Statement

Reads data from an open disk file into a variable.

Syntax

Get [#]*filenumber***,** [*recnumber*]**,** *varname*

The **Get** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
recnumber	Optional. **Variant** (**Long**). Record number (**Random** mode files) or byte number (**Binary** mode files) at which reading begins.
varname	Required. Valid variable name into which data is read.

Remarks

Data read with **Get** is usually written to a file with **Put**.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. If you omit *recnumber*, the next record or byte following the last **Get** or **Put** statement (or pointed to by the last **Seek** function) is read. You must include delimiting commas, for example:

```
Get #4,,FileBuffer
```

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** 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 the record length match the length of the data being read.

If the variable being read into is a variable-length string, **Get** reads a 2-byte descriptor containing the string length and then reads the data that goes into the variable. Therefore, 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 into is a Variant of numeric type, **Get** reads 2 bytes identifying the **VarType** of the **Variant** and then the data that goes into 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.

Note You can use the **Get** statement to read a **Variant** array from disk, but you can't use **Get** to read a scalar **Variant** containing an array. You also can't use **Get** to read objects from disk.

If the variable being read into is a **Variant** of **VarType** 8 (**String**), **Get** reads 2 bytes identifying the **VarType**, 2 bytes indicating the length of the string, and then reads 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 into is a dynamic array, **Get** reads a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. 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 read the array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk.

```
Dim MyArray(1 To 5,1 To 10) As Integer
```

The 118 bytes are distributed as follows: 18 bytes for the descriptor (2 + 8 * 2), and 100 bytes for the data (5 * 10 * 2).

If the variable being read into is a fixed-size array, **Get** reads only the data. No descriptor is read.

If the variable being read into 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 read.

Get reads elements of user-defined types as if each were being read individually, except that there is no padding between elements. On disk, a dynamic array in a user-defined type (written with **Put**) is prefixed by a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. 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 read the individual elements, including any arrays and their descriptors.

For files opened in **Binary** mode, all of the **Random** rules apply, except:

The **Len** clause in the **Open** statement has no effect. **Get** reads all variables from disk contiguously; that is, with no padding between records.

For any array other than an array in a user-defined type, **Get** reads only the data. No descriptor is read.

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 already in the string. For example, the following statements read 10 bytes from file number 1:

```
VarString = String(10," ")
Get #1,,VarString
```

GoSub...Return Statement

Branches to and returns from a subroutine within a procedure.

Syntax

GoSub *line* . . .
line
. . .

Return

The *line* argument can be any line label or line number.

Remarks

You can use **GoSub** and **Return** anywhere in a procedure, but **GoSub** and the corresponding **Return** statement must be in the same procedure. A subroutine can contain more than one **Return** statement, but the first **Return** statement encountered causes the flow of execution to branch back to the statement immediately following the most recently executed **GoSub** statement.

Note You can't enter or exit **Sub** procedures with **GoSub...Return**.

Tip Creating separate procedures that you can call may provide a more structured alternative to using **GoSub...Return**.

GoTo Statement

Branches unconditionally to a specified line within a procedure.

Syntax

GoTo *line*

The required *line* argument can be any line label or line number.

Remarks

GoTo can branch only to lines within the procedure where it appears.

Note Too many **GoTo** statements can make code difficult to read and debug. Use structured control statements (**Do...Loop**, **For...Next**, **If...Then...Else**, **Select Case**) whenever possible.

If...Then...Else Statement

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

If *condition* **Then** [*statements*] [**Else** *elsestatements*]

Or, you can use the block form syntax:

If *condition* **Then** [*statements*]

[**ElseIf** *condition-n* **Then**
  [*elseifstatements*] . . .

[**Else**
  [*elsestatements*]]

End If

The **If...Then...Else** statement syntax has these parts:

Part	Description
condition	Required. One or more of the following two types of expressions: A numeric expression or string expression that evaluates to

	True or **False**. If *condition* is Null, *condition* is treated as **False**. An expression of the form **TypeOf** *objectname* **Is** *objecttype*. The *objectname* is any object reference and *objecttype* is any valid object type. The expression is **True** if *objectname* is of the object type specified by *objecttype*; otherwise it is **False**.
statements	Optional in block form; required in single-line form that has no **Else** clause. One or more statements separated by colons; executed if *condition* is **True**.
condition-n	Optional. Same as *condition*.
elseifstatements	Optional. One or more statements executed if associated *condition-n* is **True**.
elsestatements	Optional. One or more statements executed if no previous *condition* or *condition-n* expression is **True**.

Remarks

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

Note With the single-line form, it is possible to have multiple statements executed as the result of an **If...Then** decision. All statements 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 +
```

A block form **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 preceding them. The block **If** must end with an **End If** statement.

To determine whether or not a statement is a block **If**, examine what follows the **Then** keyword. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** clauses as you want in a block **If**, but none can appear after an **Else** clause. Block **If** statements can be nested; that is, contained within one another.

When executing a block **If** (second syntax), *condition* is tested. If *condition* is **True**, the statements following **Then** are executed. If *condition* is **False**, each **ElseIf** condition (if any) is evaluated in turn. When a **True** condition is found, the statements immediately following the associated **Then** are executed. If none of the **ElseIf** conditions are **True** (or if there are no **ElseIf** clauses), the statements following **Else** are executed. After executing the statements following **Then** or **Else**, execution continues with the statement following **End If**.

Tip **Select Case** may be more useful when evaluating a single expression that has several possible actions. However, the **TypeOf** *objectname* **Is** *objecttype* clause can't be used with the **Select Case** statement.

Note **TypeOf** cannot be used with hard data types such as Long, Integer, and so forth other than Object.

Input # Statement

Reads data from an open sequential file and assigns the data to variables.

Syntax

Input #*filenumber, varlist*

The **Input #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
varlist	Required. Comma-delimited list of variables that are assigned values read from the file — can't be an array or object variable. However, variables that describe an element of an array or user-defined type may be used.

Remarks

Data read with **Input #** is usually written to a file with **Write #**. Use this statement only with files opened in **Input** or **Binary** mode.

When read, standard string or numeric data is assigned to variables without modification. The following table illustrates how other input data is treated:

Data	Value assigned to variable
Delimiting comma or blank line	Empty
#NULL#	Null
#TRUE# or #FALSE#	**True** or **False**
#yyyy-mm-dd hh:mm:ss#	The date and/or time represented by the expression
#ERROR *errornumber#*	*errornumber* (variable is a Variant tagged as an error)

Double quotation marks (`" "`) within input data are ignored.

Note You should not write strings that contain embedded quotation marks, for example, `"1,2""X"` for use with the **Input #** statement: **Input #** parses this string as two complete and separate strings.

Data items in a file must appear in the same order as the variables in *varlist* and match variables of the same data type. If a variable is numeric and the data is not numeric, a value of zero is assigned to the variable.

If you reach the end of the file while you are inputting a data item, the input is terminated and an error occurs.

Note To be able to correctly read data from a file into variables using **Input #**, use the **Write #** statement instead of the **Print #** statement to write the data to the files. Using **Write #** ensures each separate data field is properly delimited.

Kill Statement

Deletes files from a disk.

Syntax

Kill *pathname*

The required *pathname* argument is a string expression that specifies one or more file names to be deleted. The *pathname* may include the directory or folder, and the drive.

Remarks

In Microsoft Windows, **Kill** supports the use of multiple-character (*****) and single-character (**?**) wildcards to specify multiple files. However, on the Macintosh, these characters are treated as valid file name characters and can't be used as wildcards to specify multiple files.

Since the Macintosh doesn't support the wildcards, use the file type to identify groups of files to delete. You can use the **MacID** function to specify file type instead of repeating the command with separate file names. For example, the following statement deletes all TEXT files in the current folder.

```
Kill MacID("TEXT")
```

If you use the **MacID** function with **Kill** in Microsoft Windows, an error occurs.

An error occurs if you try to use **Kill** to delete an open file.

Note To delete directories, use the **RmDir** statement.

Let Statement

Assigns the value of an expression to a variable or property.

Syntax

[**Let**] *varname* = *expression*

The **Let** statement syntax has these parts:

Part	Description
Let	Optional. Explicit use of the **Let** keyword is a matter of style, but it is usually omitted.
varname	Required. Name of the variable or property; follows standard variable naming conventions.
expression	Required. Value assigned to the variable or property.

Remarks

A value expression can be assigned to a variable or property only if it is of a data type that is compatible with the variable. You can't assign string expressions to numeric variables, and you can't assign numeric expressions to string variables. If you do, an error occurs at compile time.

Variant variables can be assigned either string or numeric expressions. However, the reverse is not always true. Any **Variant** except a Null can be assigned to a string variable, but only a **Variant** whose value 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 an expression of one numeric type to a variable of a different numeric type coerces the value of the expression into the numeric 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. Use the **Set** statement to assign object references to variables.

Line Input # Statement

Reads a single line from an open sequential file and assigns it to a String variable.

Syntax

Line Input #*filenumber*, *varname*

The **Line Input #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
varname	Required. Valid Variant or **String** variable name.

Remarks

Data read with **Line Input #** is usually written from a file with **Print #**.

The **Line Input #** statement reads from a file one character at a time until it encounters a carriage return (**Chr(**13**)**) or carriage return–linefeed (**Chr(**13**)** + **Chr(**10**)**) sequence. Carriage return–linefeed sequences are skipped rather than appended to the character string.

Load Statement

Loads an object but doesn't show it.

Syntax

Load *object*

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

When an object is loaded, it is placed in memory, but isn't visible. Use the **Show** method to make the object visible. Until an object is visible, a user can't interact with it. The object can be manipulated programmatically in its Initialize event procedure.

Lock,
Unlock

Statements

Controls access by other processes to all or part of a file opened using the **Open** statement.

Syntax

Lock [#]*filenumber*[, *recordrange*]
. . .

Unlock [#]*filenumber*[, *recordrange*]

The **Lock** and **Unlock** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
recordrange	Optional. The range of records to lock or unlock.

Settings

The *recordrange* argument settings are:

recnumber | [*start*] **To** *end*

Setting	Description
recnumber	Record number (**Random** mode files) or byte number (**Binary** mode files) at which locking or unlocking begins.
start	Number of the first record or byte to lock or unlock.
end	Number of the last record or byte to lock or unlock.

Remarks

The **Lock** and **Unlock** statements are used in environments where several processes might need access to the same file.

Lock and **Unlock** statements are always used in pairs. The arguments to **Lock** and **Unlock** must match exactly.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. 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 *recnumber* locks the entire file; using **Unlock** without *recnumber* unlocks the entire file.

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*.

Caution Be sure to remove all locks with an **Unlock** statement before closing a file or quitting your program. Failure to remove locks produces unpredictable results.

LSet Statement

Left aligns a string within a string variable, or copies a variable of one user-defined type to another variable of a different user-defined type.

Syntax

LSet *stringvar* = *string*

LSet *varname1* = *varname2*

The **LSet** statement syntax has these parts:

Part	Description
stringvar	Required. Name of string variable.
string	Required. String expression to be left-aligned within *stringvar.*
varname1	Required. Variable name of the user-defined type being copied to.
varname2	Required. Variable name of the user-defined type being copied from.

Remarks

LSet replaces any leftover characters in *stringvar* with spaces.

If *string* is longer than *stringvar*, **LSet** places only the leftmost characters, up to the length of the *stringvar,* in *stringvar*.

Warning Using **LSet** to copy a variable of one user-defined type into a variable of a different user-defined type is not recommended. Copying data of one data type into space reserved for a different data type can cause unpredictable results.

When you copy a variable from one user-defined type to another, the binary data from one variable is copied into the memory space of the other, without regard for the data types specified for the elements.

Mid

Statement

Replaces a specified number of characters in a **Variant** (**String**) variable with characters from another string.

Syntax

Mid(*stringvar*, *start*[, *length*]) = *string*

The **Mid** statement syntax has these parts:

Part	Description
stringvar	Required. Name of string variable to modify.
start	Required; **Variant** (**Long**). Character position in *stringvar* where the replacement of text begins.
length	Optional; **Variant** (**Long**). Number of characters to replace. If omitted, all of *string* is used.

string Required. String expression that replaces part of *stringvar*.

Remarks

The number of characters replaced is always less than or equal to the number of characters in *stringvar*.

Note Use the **MidB** statement with byte data contained in a string. In the **MidB** statement, *start* specifies the byte position within *stringvar* where replacement begins and *length* specifies the numbers of bytes to replace.

MkDir Statement

Creates a new directory or folder.

Syntax

MkDir *path*

The required *path* argument is a string expression that identifies the directory or folder to be created. The *path* may include the drive. If no drive is specified, **MkDir** creates the new directory or folder on the current drive.

Name

Statement

Renames a disk file, directory, or folder.

Syntax

Name *oldpathname* **As** *newpathname*

The **Name** statement syntax has these parts:

Part	Description
oldpathname	Required. String expression that specifies the existing file name and location — may include directory or folder, and drive.
newpathname	Required. String expression that specifies the new file name and location — may include directory or folder, and drive. The file name specified by *newpathname* can't already exist.

Remarks

The Name statement renames a file and moves it to a different directory or folder, if necessary. Name can move a file across drives, but it can only rename an existing directory or folder when both newpathname and oldpathname are located on the same drive. Name cannot create a new file, directory, or folder.

Using **Name** on an open file produces an error. You must close an open file before renaming it. **Name** arguments cannot include multiple-character (*****) and single-character (**?**) wildcards.

On Error Statement

Enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine.

Syntax

On Error GoTo *line*

On Error Resume Next

On Error GoTo 0

The **On Error** statement syntax can have any of the following forms:

Statement	Description
On Error GoTo *line*	Enables the error-handling routine that starts at *line* specified in the required *line* argument. The *line* argument is any line label or line number. If a run-time error occurs, control branches to *line*, making the error handler active. The specified *line* must be in the same procedure as the **On Error** statement; otherwise, a compile-time error occurs.
On Error Resume Next	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred where execution continues.

	Use this form rather than **On Error GoTo** when accessing objects.
On Error GoTo 0	Disables any enabled error handler in the current procedure.

Remarks

If you don't use an **On Error** statement, any run-time error that occurs is fatal; that is, an error message is displayed and execution stops.

An "enabled" error handler is one that is turned on by an **On Error** statement; an "active" error handler is an enabled handler that is in the process of handling an error. If an error occurs while an error handler is active (between the occurrence of the error and a **Resume**, **Exit Sub**, **Exit Function**, or **Exit Property** statement), the current procedure's error handler can't handle the error. Control returns to the calling procedure. If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no inactive, enabled 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 a calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the **Resume** statement.

Note An error-handling routine is not a **Sub** procedure or **Function** procedure. It is a section of code marked by a line label or line number.

Error-handling routines rely on the value in the **Number** property of the **Err** object to determine the cause of the error. The error-handling routine should test or save relevant property values in the **Err** object before any other error can occur or before a procedure that might cause an error is called. The property values in the **Err** object reflect only the most recent error. The error message associated with **Err.Number** is contained in **Err.Description**.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure

containing the **On Error Resume Next** statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur, rather than transferring control to another location within the procedure. An **On Error Resume Next** statement becomes inactive when another procedure is called, so you should execute an **On Error Resume Next** statement in each called routine if you want inline error handling within that routine.

Note The **On Error Resume Next** construct may be preferable to **On Error GoTo** when handling errors generated during access to other objects. Checking **Err** after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in **Err.Number**, as well as which object originally generated the error (the object specified in **Err.Source**).

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 running when no error has occurred, place an **Exit Sub**, **Exit Function**, or **Exit Property** statement immediately before the error-handling routine, as in the following fragment:

```
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 separate it from the procedure flow. Error-handling code can be placed anywhere in a procedure.

Untrapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, untrapped errors are only returned to the controlling application if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle errors passed back from them unhandled. If you cannot handle such errors, map the error code in **Err.Number** to one of your own errors, and then pass them back to the caller of your object. You should specify your error by adding your error code to the **vbObjectError** constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = vbObjectError + 1052
```

Note System errors during calls to Windows dynamic-link libraries (DLL) or Macintosh code resources do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the **Err** object's **LastDLLError** property. **LastDLLError** always returns zero on the Macintosh.

On...GoSub, On...GoTo Statements

Branch to one of several specified lines, depending on the value of an expression.

Syntax

On *expression* **GoSub** *destinationlist*

On *expression* **GoTo** *destinationlist*

The **On...GoSub** and **On...GoTo** statement syntax has these parts:

Part	Description
expression	Required. Any numeric expression that evaluates to a whole number between 0 and 255, inclusive. If *expression* is any number other than a whole number, it is rounded before it is evaluated.
destinationlist	Required. List of line numbers or line labels separated by commas.

Remarks

The value of *expression* determines which line is branched to in *destinationlist*. 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:

If *expression* is	Then
Equal to 0	Control drops to the statement following **On...GoSub** or **On...GoTo**.
Greater than number of items in list	Control drops to the statement following **On...GoSub** or **On...GoTo**.
Negative	An error occurs.
Greater than 255	An error occurs.

You can mix line numbers and line labels in the same list. You can use as many line labels and line numbers as you like with **On...GoSub** and **On...GoTo**. However, if you use more labels or numbers than fit on a single line, you must use the line-continuation character to continue the logical line onto the next physical line.

Tip **Select Case** provides a more structured and flexible way to perform multiple branching.

Open Statement

Enables input/output (I/O) to a file.

Syntax

Open *pathname* **For** *mode* [**Access** *access*] [*lock*] **As** [#]*filenumber*
[**Len**=*reclength*]

The **Open** statement syntax has these parts:

Part	Description

pathname	Required. String expression that specifies a file name — may include directory or folder, and drive.
mode	Required. Keyword specifying the file mode: **Append**, **Binary**, **Input**, **Output**, or **Random**. If unspecified, the file is opened for **Random** access.
access	Optional. Keyword specifying the operations permitted on the open file: **Read**, **Write**, or **Read Write**.
lock	Optional. Keyword specifying the operations restricted on the open file by other processes: **Shared**, **Lock Read**, **Lock Write**, and **Lock Read Write**.
filenumber	Required. A valid file number in the range 1 to 511, inclusive. Use the **FreeFile** function to obtain the next available file number.
reclength	Optional. Number less than or equal to 32,767 (bytes). For files opened for random access, this value is the record length. For sequential files, this value is the number of characters buffered.

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 to use with the buffer.

If the file specified by *pathname* doesn't exist, it is created when a file is opened for **Append**, **Binary**, **Output**, or **Random** modes.

If the file is already opened by another process and the specified type of access is not allowed, the **Open** operation fails and an error occurs.

The **Len** clause is ignored if *mode* is **Binary**.

Important In **Binary**, **Input**, and **Random** modes, you can open a file using a different file number without first closing the file. In **Append** and **Output** modes, you must close a file before opening it with a different file number.

Option Base Statement

Used at module level to declare the default lower bound for array subscripts.

Syntax

Option Base {**0** | **1**}

Remarks

Because the default base is **0**, the **Option Base** statement is never required. If used, the statement must appear in a module before any procedures. **Option Base** can appear only once in a module and must precede array declarations that include dimensions.

Note The **To** clause in the **Dim**, **Private**, **Public**, **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. The base of an array created with the the **ParamArray** keyword is zero; **Option Base** does not affect **ParamArray** (or the **Array** function, when qualified with the name of its type library, for example **VBA.Array**).

The **Option Base** statement only affects the lower bound of arrays in the module where the statement is located.

Option Compare Statement

Used at module level to declare the default comparison method to use when string data is compared.

Syntax

Option Compare {**Binary** | **Text** | **Database**}

Remarks

If used, the **Option Compare** statement must appear in a module before any procedures.

The **Option Compare** statement specifies the string comparison method (**Binary**, **Text**, or **Database**) for a module. If a module doesn't include an **Option Compare** statement, the default text comparison method is **Binary**.

Option Compare Binary results in string comparisons based on a sort order derived from the internal binary representations of the characters. In Microsoft Windows, sort order is determined by the code page. A typical binary sort order is shown in the following example:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à <
```

Option Compare Text results in string comparisons based on a case-insensitive text sort order determined by your system's locale. When the same characters are sorted using **Option Compare Text**, the following text sort order is produced:

```
(A=a) < ( À=à) < (B=b) < (E=e) < (Ê=ê) < (Z=z)
```

Option Compare Database can only be used within Microsoft Access. This results in string comparisons based on the sort order determined by the locale ID of the database where the string comparisons occur.

Option Explicit Statement

Used at module level to force explicit declaration of all variables in that module.

Syntax

Option Explicit

Remarks

If used, the **Option Explicit** statement must appear in a module before any procedures.

When **Option Explicit** appears in a module, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, **ReDim**, or **Static** statements. If you attempt to use an undeclared variable name, an error occurs at compile time.

If you don't use the **Option Explicit** statement, all undeclared variables are of **Variant** type unless the default type is otherwise specified with a **Def***type* statement.

Note Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.

Option Private Statement

When used in host applications that allow references across multiple projects, **Option Private Module** prevents a module's contents from being referenced outside its project. In host applications that don't permit such references, for example, standalone versions of Visual Basic, **Option Private** has no effect.

Syntax

Option Private Module

Remarks

If used, the **Option Private** statement must appear at module level, before any procedures.

When a module contains **Option Private Module**, the public parts, for example, variables, objects, and user-defined types declared at module level, are still available within the project containing the module, but they are not available to other applications or projects.

Note **Option Private** is only useful for host applications that support simultaneous loading of multiple projects and permit references between the loaded projects. For example, Microsoft Excel permits loading of multiple projects and **Option Private Module** can be used to restrict cross-project visibility. Although Visual Basic permits loading of multiple projects, references

between projects are never permitted in Visual Basic.

Print # Statement

Writes display-formatted data to a sequential file.

Syntax

Print #*filenumber*, [*outputlist*]

The **Print #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
outputlist	Optional. Expression or list of expressions to print.

Settings

The *outputlist* argument settings are:

[{**Spc(***n***)** | **Tab**[**(***n***)**]}] [*expression*] [*charpos*]

Setting	Description
Spc(*n***)**	Used to insert space characters in the output, where *n* is the number of space characters to insert.
Tab(*n***)**	Used to position the insertion point to an absolute column number, where *n* is the column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone.
expression	Numeric expressions or string expressions to print.
charpos	Specifies the insertion point for the next character. Use a semicolon to position the insertion point immediately after the last character displayed. Use **Tab(***n***)** to position the insertion point to an absolute column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone. If *charpos* is omitted, the next character is printed on the next line.

Remarks

Data written with **Print #** is usually read from a file with **Line Input #** or **Input**.

If you omit *outputlist* and include only a list separator after *filenumber*, a blank line is printed to the file. Multiple expressions can be separated with either a space or a semicolon. A space has the same effect as a semicolon.

For Boolean data, either `True` or `False` is printed. The **True** and **False** keywords are not translated, regardless of the locale.

Date data is written to the file using the standard short date format recognized by your system. When either the date or the time component is missing or zero, only the part provided gets written to the file.

Nothing is written to the file if *outputlist* data is Empty. However, if *outputlist* data is Null, **Null** is written to the file.

For **Error** data, the output appears as `Error errorcode`. The **Error** keyword is not translated regardless of the locale.

All data written to the file using **Print #** is internationally aware; that is, the data is properly formatted using the appropriate decimal separator.

Because **Print #** writes an image of the data to the file, you must delimit the data so it prints correctly. If you use **Tab** with no arguments to move the print position to the next print zone, **Print #** also writes the spaces between print fields to the file.

Note If, at some future time, you want to read the data from a file using the **Input #** statement, use the **Write #** statement instead of the **Print #** statement to write the data to the file. Using **Write #** ensures the integrity of each separate data field by properly delimiting it, so it can be read back in using **Input #**. Using **Write #** also ensures it can be correctly read in any locale.

Private Statement

Used at module level to declare private variables and allocate storage space.

Syntax

Private [**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*] [**,**[**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*]] **. . .**

The **Private** statement syntax has these parts:

Part	Description

WithEvents	Optional. Keyword that specifies that *varname* is an object variable used to respond to events triggered by an ActiveX object. **WithEvents** is valid only in class modules. You can declare as many individual variables as you like using **WithEvents**, but you can't create arrays with **WithEvents**. You can't use **New** with **WithEvents**.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,**[*lower* **To**] *upper*] **. . .** When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type, can't be used to declare instances of dependent objects, and can’t be used with **WithEvents**.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined.

Remarks

Private variables are available only to the module in which they are declared.

Use the **Private** statement to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**:

```
Private NumberOfEmployees As Integer
```

You can also use a **Private** statement to declare the object type of a variable. The following statement declares a variable for a new instance of a worksheet.

```
Private X As New Worksheet
```

If the **New** keyword isn't used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it's assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

If you don't specify a data type or object type, and there is no **Def***type* statement in the module, the variable is **Variant** by default.

You can also use the **Private** statement with empty parentheses to declare a dynamic array. 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 was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Note When you use the **Private** statement in a procedure, you generally put the **Private** statement at the beginning of the procedure.

Property Get Statement

Declares the name, arguments, and code that form the body of a **Property** procedure, which gets the value of a property.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Property Get** *name* [**(***arglist***)**] [**As** *type*]
  [*statements*]
  [*name* = *expression*]
  [**Exit Property**]
  [*statements*]
  [*name* = *expression*]

End Property

The **Property Get** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Property Get** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Get** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Property Get** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Get** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property**

	Get procedure, even if they are used in the procedure.
name	Required. Name of the **Property Get** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Let** or **Property Set** procedure in the same module.
arglist	Optional. List of variables representing arguments that are passed to the **Property Get** procedure when it is called. Multiple arguments are separated by commas. The name and data type of each argument in a **Property Get** procedure must be the same as the corresponding argument in a **Property Let** procedure (if one exists).
type	Optional. Data type of the value returned by the **Property Get** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (except fixed length), Object, Variant, user-defined type, and Arrays. The return *type* of a **Property Get** procedure must be the same data type as the last (or sometimes the only) argument in a corresponding **Property Let** procedure (if one exists) that defines the value assigned to the property on the right side of an expression.
statements	Optional. Any group of statements to be executed within the body of the **Property Get** procedure.
expression	Optional. Value of the property returned by the procedure defined by the **Property Get** statement.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword.

ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported), **Date**, **String** (variable length only), **Object**, **Variant**, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Property** procedures are public by default. If **Static** is not used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

All executable code must be in procedures. You can't define a **Property Get** procedure inside another **Property**, **Sub**, or **Function** procedure.

The **Exit Property** statement causes an immediate exit from a **Property Get** procedure. Program execution continues with the statement following the statement that called the **Property Get** procedure. Any number of **Exit Property**

statements can appear anywhere in a **Property Get** procedure.

Like a **Sub** and **Property Let** procedure, a **Property Get** 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** or **Property Let** procedure, you can use a **Property Get** procedure on the right side of an expression in the same way you use a **Function** or a property name when you want to return the value of a property.

Property Let Statement

Declares the name, arguments, and code that form the body of a **Property Let** procedure, which assigns a value to a property.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Property Let** *name* **(**[*arglist***,**] *value***)**
  [*statements*]
  [**Exit Property**]
  [*statements*]

End Property

The **Property Let** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Property Let** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Let** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Property Let** procedure is visible throughout the project, but

	not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Let** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property Let** procedure, even if they are used in the procedure.
name	Required. Name of the **Property Let** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Set** procedure in the same module.
arglist	Required. List of variables representing arguments that are passed to the **Property Let** procedure when it is called. Multiple arguments are separated by commas. The name and data type of each argument in a **Property Let** procedure must be the same as the corresponding argument in a **Property Get** procedure.
value	Required. Variable to contain the value to be assigned to the property. When the procedure is called, this argument appears on the right side of the calling expression. The data type of *value* must be the same as the return type of the corresponding **Property Get** procedure.
statements	Optional. Any group of statements to be executed within the **Property Let** procedure.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. Note that it is not possible for the right side of a **Property Let** expression to be **Optional**.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference.

	ByRef is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), Object, Variant, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Note Every **Property Let** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual value to be assigned to the property when the procedure defined by the **Property Let** statement is invoked. That argument is referred to as *value* in the preceding syntax.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Property** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

All executable code must be in procedures. You can't define a **Property Let** procedure inside another **Property**, **Sub**, or **Function** procedure.

The **Exit Property** statement causes an immediate exit from a **Property Let**

procedure. Program execution continues with the statement following the statement that called the **Property Let** procedure. Any number of **Exit Property** statements can appear anywhere in a **Property Let** procedure.

Like a **Function** and **Property Get** procedure, a **Property Let** 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** and **Property Get** procedure, both of which return a value, you can only use a **Property Let** procedure on the left side of a property assignment expression or **Let** statement.

Property Set Statement

Declares the name, arguments, and code that form the body of a **Property** procedure, which sets a reference to an object.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Property Set** *name* **(**[*arglist***,**] *reference***)**
  [*statements*]
  [**Exit Property**]
  [*statements*]

End Property

The **Property Set** statement syntax has these parts:

Part	Description
Optional	Optional. Indicates that the argument may or may not be supplied by the caller.
Public	Optional. Indicates that the **Property Set** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Set** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Property Set** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Set** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property Set** procedure, even if they are used in the procedure.

name	Required. Name of the **Property Set** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Let** procedure in the same module.
arglist	Required. List of variables representing arguments that are passed to the **Property Set** procedure when it is called. Multiple arguments are separated by commas.
reference	Required. Variable containing the object reference used on the right side of the object reference assignment.
statements	Optional. Any group of statements to be executed within the body of the **Property** procedure.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. Note that it is not possible for the right side of a **Property Set** expression to be **Optional**.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure;

	may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), Object, Variant, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Note Every **Property Set** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual object reference for the property when the procedure defined by the **Property Set** statement is invoked. It is referred to as *reference* in the preceding syntax. It can't be **Optional**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Property** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

All executable code must be in procedures. You can't define a **Property Set** procedure inside another **Property**, **Sub**, or **Function** procedure.

The **Exit Property** statement causes an immediate exit from a **Property Set** procedure. Program execution continues with the statement following the statement that called the **Property Set** procedure. Any number of **Exit Property** statements can appear anywhere in a **Property Set** procedure.

Like a **Function** and **Property Get** procedure, a **Property Set** 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** and **Property Get** procedure, both of which return a value, you can only use a **Property Set** procedure on the left side of an object reference assignment (**Set** statement).

Public Statement

Used at module level to declare public variables and allocate storage space.

Syntax

Public [**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*] [**,**[**WithEvents**] *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*]] **. . .**

The **Public** statement syntax has these parts:

Part	Description
WithEvents	Optional. Keyword specifying that *varname* is an object variable used to respond to events triggered by an ActiveX object. **WithEvents** is valid only in class modules. You can declare as many individual variables as you like using **WithEvents**, but you can't create arrays with **WithEvents**. You can't use **New** with **WithEvents**.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,**[*lower* **To**] *upper*] **. . .** When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type, can't be used to declare instances of dependent objects, and can't be used with **WithEvents**.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined.

Remarks

Variables declared using the **Public** statement are available to all procedures in all modules in all applications unless **Option Private Module** is in effect; in which case, the variables are public only within the project in which they reside.

Caution The **Public** statement can't be used in a class module to declare a fixed-length string variable.

Use the **Public** statement to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**:

```
Public NumberOfEmployees As Integer
```

Also use a **Public** statement to declare the object type of a variable. The following statement declares a variable for a new instance of a worksheet.

```
Public X As New Worksheet
```

If the **New** keyword is not used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

You can also use the **Public** statement with empty parentheses to declare a dynamic array. 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 was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.

If you don't specify a data type or object type and there is no **Def***type* statement in the module, the variable is **Variant** by default.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Put Statement

Writes data from a variable to a disk file.

Syntax

Put [#]*filenumber*, [*recnumber*], *varname*

The **Put** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
recnumber	Optional. **Variant** (**Long**). Record number (**Random** mode files) or byte number (**Binary** mode files) at which writing begins.
varname	Required. Name of variable containing data to be written to disk.

Remarks

Data written with **Put** is usually read from a file with **Get**.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. If you omit *recnumber*, the next record or byte after the last **Get** or **Put** statement or pointed to by the last **Seek** function is written. You must include delimiting commas, for example:

```
Put #4,,FileBuffer
```

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** 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 the record length match the length of the data being written. If the length of the data being written is greater than the length specified in the **Len** clause of the **Open** statement, an error occurs.

If the variable being written is a variable-length string, **Put** writes a 2-byte descriptor containing the 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 Variant of a numeric type, **Put** writes 2 bytes identifying the **VarType** of the **Variant** and then writes 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.

Note You can use the **Put** statement to write a **Variant** array to disk, but you can't use **Put** to write a scalar **Variant** containing an array to disk. You also can't use **Put** to write objects to disk.

If the variable being written is a **Variant** of **VarType** 8 (**String**), **Put** writes 2 bytes identifying the **VarType**, 2 bytes indicating the length of the string, and then writes 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 a dynamic array, **Put** writes a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. 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 array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk.

```
Dim MyArray(1 To 5,1 To 10) As Integer
```

The 118 bytes are distributed as follows: 18 bytes for the descriptor (2 + 8 * 2), and 100 bytes for the data (5 * 10 * 2).

If the variable being written is a fixed-size array, **Put** writes only the data. No descriptor is written to disk.

If the variable being written is any other type of variable (not a variable-

length string or 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.

Put writes elements of user-defined types as if each were written individually, except there is no padding between elements. On disk, a dynamic array in a user-defined type written with **Put** is prefixed by a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. 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, including any arrays and their descriptors.

For files opened in **Binary** mode, all of the **Random** rules apply, except:

The **Len** clause in the **Open** statement has no effect. **Put** writes all variables to disk contiguously; that is, with no padding between records.

For any array other than an array in a user-defined type, **Put** writes only the data. No descriptor is written.

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$(10," ")
Put #1,,VarString$
```

Initializes the random-number generator.

Syntax

Randomize [*number*]

The optional *number* argument is a Variant or any valid numeric expression.

Remarks

Randomize uses *number* to initialize the **Rnd** function's random-number generator, giving it a new seed value. If you omit *number,* the value returned by the system timer is used as the new seed value.

If **Randomize** is not used, the **Rnd** function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

Note To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.

ReDim Statement

Used at procedure level to reallocate storage space for dynamic array variables.

Syntax

ReDim [**Preserve**] *varname***(***subscripts***)** [**As** *type*] [**,** *varname***(***subscripts***)** [**As** *type*]] **. . .**

The **ReDim** statement syntax has these parts:

Part	Description
Preserve	Optional. Keyword used to preserve the data in an existing array when you change the size of the last dimension.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Required. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,**[*lower* **To**] *upper*] **. . .** When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined. For a **Variant** containing an array, *type* describes the type of each element of the array, but doesn't change the **Variant** to some other type.

Remarks

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a **Private**, **Public**, or **Dim** statement with empty parentheses (without dimension subscripts).

You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array. However, you can't declare an array of one data type and later use **ReDim** to change the array to another data type, unless the array is contained in a **Variant**. If the array is contained in a **Variant**, the type of the

elements can be changed using an **As** *type* clause, unless you're using the **Preserve** keyword, in which case, no changes of data type are permitted.

If you use the **Preserve** keyword, you can resize only the last array dimension and you can't change the number of dimensions at all. 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, 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)
```

Similarly, when you use **Preserve**, you can change the size of the array only by changing the upper bound; changing the lower bound causes an error.

If you make an array smaller than it was, data in the eliminated elements will be lost. If you pass an array to a procedure by reference, you can't redimension the array within the procedure.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable. A variable that refers to an object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

Caution The **ReDim** statement acts as a declarative statement if the variable it declares doesn't exist at module level or procedure level. If another variable with the same name is created later, even in a wider scope, **ReDim** will refer to the later variable and won't necessarily cause a compilation error, even if **Option Explicit** is in effect. To avoid such conflicts, **ReDim** should not be used as a declarative statement, but simply for redimensioning arrays.

Note To resize an array contained in a **Variant**, you must explicitly declare the **Variant** variable before attempting to resize its array.

Rem Statement

Used to include explanatory remarks in a program.

Syntax

Rem *comment*

You can also use the following syntax:

' *comment*

The optional *comment* argument is the text of any comment you want to include. A space is required between the **Rem** keyword and *comment*.

Remarks

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. If the **Rem** keyword follows other statements on a line, it must be separated from the statements by a colon (**:**).

You can use an apostrophe (') instead of the **Rem** keyword. When you use an apostrophe, the colon is not required after other statements.

Closes all disk files opened using the **Open** statement.

Syntax

Reset

Remarks

The **Reset** statement closes all active files opened by the **Open** statement and writes the contents of all file buffers to disk.

Resume Statement

Resumes execution after an error-handling routine is finished.

Syntax

Resume [**0**]

Resume Next

Resume *line*

The **Resume** statement syntax can have any of the following forms:

Statement	Description
Resume	If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.
Resume Next	If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or **On Error Resume Next** statement).
Resume *line*	Execution resumes at *line* specified in the required *line* argument. The *line* argument is a line label or line number and must be in the same procedure as the error handler.

Remarks

If you use a **Resume** statement anywhere except in an error-handling routine, an error occurs.

Removes an existing directory or folder.

Syntax

RmDir *path*

The required *path* argument is a string expression that identifies the directory or folder to be removed. The *path* may include the drive. If no drive is specified, **RmDir** removes the directory or folder on the current drive.

Remarks

An error occurs if you try to use **RmDir** on a directory or folder containing files. Use the **Kill** statement to delete all files before attempting to remove a directory or folder.

RSet Statement

Right aligns a string within a string variable.

Syntax

RSet *stringvar* = *string*

The **RSet** statement syntax has these parts:

Part	Description

stringvar	Required. Name of string variable.
string	Required. String expression to be right-aligned within *stringvar*.

Remarks

If *stringvar* is longer than *string*, **RSet** replaces any leftover characters in *stringvar* with spaces, back to its beginning.

Note **RSet** can't be used with user-defined types.

SaveSetting Statement

Saves or creates an application entry in the application's entry in the Windows registry or (on the Macintosh) information in the application’s initialization file.

Syntax

SaveSetting *appname, section,* *key, setting*

The **SaveSetting** statement syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project to which the setting applies. On the Macintosh, this is the filename of the initialization file in the Preferences folder in the System folder.
section	Required. String expression containing the name of the section where the key setting is being saved.
key	Required. String expression containing the name of the key setting being saved.
setting	Required. Expression containing the value that ***key*** is being set to.

Remarks

An error occurs if the key setting can’t be saved for any reason.

Sets the position for the next read/write operation within a file opened using the **Open** statement.

Syntax

Seek [#]*filenumber*, *position*

The **Seek** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
position	Required. Number in the range 1 – 2,147,483,647, inclusive, that indicates where the next read/write operation should occur.

Remarks

Record numbers specified in **Get** and **Put** statements override file positioning performed by **Seek**.

Performing a file-write operation after 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.

Select Case Statement

Executes one of several groups of statements, depending on the value of an expression.

Syntax

Select Case *testexpression* [**Case** *expressionlist-n*
[*statements-n*]] . . .
[**Case Else**
[*elsestatements*]]

End Select

The **Select Case** statement syntax has these parts:

Part	Description
testexpression	Required. Any numeric expression or string expression.
expressionlist-n	Required if a **Case** appears. Delimited list of one or more of the following forms: *expression*, *expression* **To** *expression*, **Is** *comparisonoperator expression*. The **To** keyword specifies a range of values. If you use the **To** keyword, the smaller value must appear before **To**. Use the **Is** keyword with comparison operators (except **Is** and **Like**) to specify a range of values. If not supplied, the **Is** keyword is automatically inserted.
statements-n	Optional. One or more statements executed if *testexpression* matches any part of *expressionlist-n.*
elsestatements	Optional. One or more statements executed if *testexpression* doesn't match any of the **Case** clause.

Remarks

If *testexpression* matches any **Case** *expressionlist* expression, the *statements* following that **Case** clause are executed up to the next **Case** clause, or, for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If *testexpression* matches an *expressionlist* expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the *elsestatements* to be executed if no match is found between the *testexpression* and an *expressionlist* in 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. If no **Case** *expressionlist* matches *testexpression* and there is no **Case Else** statement, 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
```

Note The **Is** comparison operator is not the same as the **Is** keyword used in the **Select Case** statement.

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 alphabetic order, and the current value of `TestItem`:

```
Case "everything", "nuts" To "soup", TestItem
```

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

SendKeys Statement

Sends one or more keystrokes to the active window as if typed at the keyboard.

Syntax

SendKeys ***string***[, ***wait***]

The **SendKeys** statement syntax has these named arguments:

Part	Description
string	Required. String expression specifying the keystrokes to send.
Wait	Optional. Boolean value specifying the wait mode. If **False** (default), control is returned to the procedure immediately after the keys are sent. If **True**, keystrokes must be processed before control is returned to the procedure.

Remarks

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 ***string***. 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 ***string***.

The plus sign (**+**), caret (**^**), percent sign (**%**), tilde (**~**), and parentheses **( )** have special meanings to **SendKeys**. To specify one of these characters, enclose it within braces (`{}`). For example, to specify the plus sign, use `{+}`. Brackets ([ ]) have no special meaning to **SendKeys**, but you must enclose them in braces. In other applications, brackets do have a special meaning that may be significant when dynamic data exchange (DDE) occurs. To specify 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
BACKSPACE	`{BACKSPACE}`, `{BS}`, or `{BKSP}`
BREAK	`{BREAK}`
CAPS LOCK	`{CAPSLOCK}`
DEL or DELETE	`{DELETE}` or `{DEL}`
DOWN ARROW	`{DOWN}`

END	{END}
ENTER	{ENTER}or ~
ESC	{ESC}
HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
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 the SHIFT, CTRL, and ALT keys, precede the key code with one or more of the following codes:

Key	Code
SHIFT	+
CTRL	^
ALT	%

To specify that any combination of SHIFT, CTRL, and ALT should be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify to hold down SHIFT while E and C are pressed, use "+(EC)". To specify to hold down SHIFT while E is pressed, followed by C 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.

Note You can't use **SendKeys** to send keystrokes to an application that is not designed to run in Microsoft Windows or Macintosh. **Sendkeys** also can't send the PRINT SCREEN key {PRTSC} to any application.

Set Statement

Assigns an object reference to a variable or property.

Syntax

Set *objectvar* = {[**New**] *objectexpression* | **Nothing**}

The **Set** statement syntax has these parts:

Part	Description
objectvar	Required. Name of the variable or property; follows standard variable naming conventions.
New	Optional. **New** is usually used during declaration to enable implicit object creation. When **New** is used with **Set**, it creates a new instance of the class. If *objectvar* contained a reference to an object, that reference is released when the new one is assigned. The **New** keyword can't be used to create new instances of any intrinsic data type and can't be used to create dependent objects.
objectexpression	Required. Expression consisting of the name of an object, another declared variable of the same object type, or a function or method that returns an object of the same object type.
Nothing	Optional. Discontinues association of *objectvar* with any

specific object. Assigning **Nothing** to *objectvar* releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.

Remarks

To be valid, *objectvar* must be an object type consistent with the object being assigned to it.

The **Dim**, **Private**, **Public**, **ReDim**, and **Static** statements only declare a variable that refers to an object. No actual object is referred to until you use the **Set** statement to assign a specific object.

The following example illustrates how **Dim** is used to declare an array with the type `Form1`. No instance of `Form1` actually exists. **Set** then assigns references to new instances of `Form1` to the `myChildForms` variable. Such code might be used to create child forms in an MDI application.

```
Dim myChildForms(1 to 4) As Form1
Set myChildForms(1) = New Form1
Set myChildForms(2) = New Form1
Set myChildForms(3) = New Form1
Set myChildForms(4) = New Form1
```

Generally, 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 such variables are references to the object rather than copies of the object, any change in the object is reflected in all variables that refer to it. However, when you use the **New** keyword in the **Set** statement, you are actually creating an instance of the object.

SetAttr Statement

Sets attribute information for a file.

Syntax

SetAttr *pathname, attributes*

The **SetAttr** statement syntax has these named arguments:

Part	Description
pathname	Required. String expression that specifies a file name — may include directory or folder, and drive.
attributes	Required. Constant or numeric expression, whose sum specifies file attributes.

Settings

The ***attributes*** argument settings are:

Constant	Value	Description
vbNormal	0	Normal (default).
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file. Not available on the Macintosh.
vbArchive	32	File has changed since last backup.
vbAlias	64	Specified file name is an alias. Available only on the Macintosh.

Note These constants are specified by Visual Basic for Applications. The names can be used anywhere in your code in place of the actual values.

Remarks

A run-time error occurs if you try to set the attributes of an open file.

Static Statement

Used at procedure level to declare variables and allocate storage space. Variables declared with the **Static** statement retain their values as long as the code is running.

Syntax

Static *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*] [**,** *varname*[**(**[*subscripts*]**)**] [**As** [**New**] *type*]] **. . .**

The **Static** statement syntax has these parts:

Part	Description
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [,[*lower* **To**] *upper*] **. . .** When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type and can't be used to declare instances of dependent objects.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, (for variable-length strings), **String** *

length (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable being defined.

Remarks

Once module code is running, variables declared with the **Static** statement retain their value until the module is reset or restarted. In class modules, variables declared with the **Static** statement retain their value in each class instance until that instance is destroyed. In form modules, static variables retain their value until the form is closed. Use the **Static** statement in nonstatic procedures to explicitly declare variables that are visible only within the procedure, but whose lifetime is the same as the module in which the procedure is defined.

Use a **Static** statement within a procedure to declare the data type of a variable that retains its value between procedure calls. For example, the following statement declares a fixed-size array of integers:

```
Static EmployeeNumber(200) As Integer
```

The following statement declares a variable for a new instance of a worksheet:

```
Static X As New Worksheet
```

If the **New** keyword isn't used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object. When you use the **New** keyword in the declaration, an instance of the object is created on the first reference to the object.

If you don't specify a data type or object type, and there is no **Def***type* statement in the module, the variable is **Variant** by default.

Note The **Static** statement and the **Static** keyword are similar, but used for different effects. If you declare a procedure using the **Static** keyword (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 for 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 specific variables within nonstatic procedures to preserve their value for as long as the program is running.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Note When you use **Static** statements within a procedure, put them at the beginning of the procedure with other declarative statements such as **Dim**.

Stop Statement

Suspends execution.

Syntax

Stop

Remarks

You can place **Stop** statements anywhere in procedures to suspend execution. Using the **Stop** statement is similar to setting a breakpoint in the code.

The **Stop** statement suspends execution, but unlike **End**, it doesn't close any files or clear variables, unless it is in a compiled executable (.exe) file.

Sub Statement

Declares the name, arguments, and code that form the body of a **Sub** procedure.

Syntax

[**Private** | **Public** | **Friend**] [**Static**] **Sub** *name* [**(***arglist***)**]
  [*statements*]
  [**Exit Sub**]
  [*statements*]

End Sub

The **Sub** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Sub** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Sub** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Sub** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. 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**, even if they are used in the procedure.
name	Required. Name of the **Sub**; follows standard variable naming conventions.
arglist	Optional. List of variables representing arguments that are passed to the **Sub** procedure when it is called. Multiple variables are separated by commas.
statements	Optional. Any group of statements to be executed within the **Sub** procedure.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Keyword indicating that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.

ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. **ParamArray** can't be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable-length only), Object, Variant, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Sub** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

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** keyword usually is not used with recursive **Sub** procedures.

All executable code must be in procedures. You can't define a **Sub** procedure inside another **Sub**, **Function**, or **Property** procedure.

The **Exit Sub** keywords cause 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.

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 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.

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. Variables that are 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 naming conflict can occur if anything you defined at the module level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant or variable, it is assumed that your procedure is referring to that module-level name. To avoid this kind of conflict, explicitly declare variables. 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.

Sets the system time.

Syntax

Time = *time*

The required *time* argument is any numeric expression, string expression, or any combination, that can represent a time.

Remarks

If *time* is a string, **Time** attempts to convert it to a time using the time separators you specified for your system. If it can't be converted to a valid time, an error occurs.

Type Statement

Used at module level to define a user-defined data type containing one or more elements.

Syntax

[**Private** | **Public**] **Type** *varname*
  elementname [**(**[*subscripts*]**)**] **As** *type*
  [*elementname* [**(**[*subscripts*]**)**] **As** *type*]
  . . .

End Type

The **Type** statement syntax has these parts:

Part	Description
Public	Optional. Used to declare user-defined types that are available to all procedures in all modules in all projects.
Private	Optional. Used to declare user-defined types that are available only within the module where the declaration is made.
varname	Required. Name of the user-defined type; follows standard variable naming conventions.
elementname	Required. Name of an element of the user-defined type. Element names also follow standard variable naming conventions, except that keywords can be used.
subscripts	When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
type	Required. Data type of the element; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings),

String * *length* (for fixed-length strings), Object, Variant, another user-defined type, or an object type.

Remarks

The **Type** statement can be used only at module level. Once you have declared a user-defined type using the **Type** statement, you can declare a variable of that type anywhere within the scope of the declaration. Use **Dim**, **Private**, **Public**, **ReDim**, or **Static** to declare a variable of a user-defined type.

In standard modules and class modules, user-defined types are public by default. This visibility can be changed using the **Private** keyword.

Line numbers and line labels aren't allowed in **Type...End Type** blocks.

User-defined types are often used with data records, which frequently consist of a number of related elements of different data types.

The following example shows the use of fixed-size arrays in a user-defined type:

```
Type StateData
    CityCode (1 To 100) As Integer    ' Declare
    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 fixed-size array within a user-defined type, its dimensions must be declared with numeric literals or constants rather than variables.

Executes a series of statements as long as a given condition is **True**.

Syntax

While *condition* [*statements*]

Wend

The **While...Wend** statement syntax has these parts:

Part	Description
condition	Required. Numeric expression or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	Optional. One or more statements executed while condition is **True**.

Remarks

If *condition* is **True**, all *statements* 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**.

Tip The **Do...Loop** statement provides a more structured and flexible way to perform looping.

Width #

Statement

Assigns an output line width to a file opened using the **Open** statement.

Syntax

Width #*filenumber*, *width*

The **Width #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
width	Required. Numeric expression in the range 0–255, 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 *width* is 0.

With Statement

Executes a series of statements on a single object or a user-defined type.

Syntax

With *object* [*statements*]

End With

The **With** statement syntax has these parts:

Part	Description
object	Required. Name of an object or a user-defined type.
statements	Optional. One or more statements to be executed on *object*.

Remarks

The **With** statement allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the **With** control structure, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of the **With** statement to assign values to several properties of the same object.

```
With MyLabel
    .Height = 2000
    .Width = 2000
    .Caption = "This is MyLabel"
End With
```

Note Once a **With** block is entered, *object* can't be changed. As a result, you can't use a single **With** statement to affect a number of different objects.

You can nest **With** statements by placing one **With** block within another. However, because members of outer **With** blocks are masked within the inner **With** blocks, you must provide a fully qualified object reference in an inner **With** block to any member of an object in an outer **With** block.

Note In general, it's recommended that you don't jump into or out of **With** blocks. If statements in a **With** block are executed, but either the **With** or **End With** statement is not executed, a temporary variable containing a reference to the object remains in memory until you exit the procedure.

Write # Statement

Writes data to a sequential file.

Syntax

Write #*filenumber*, [*outputlist*]

The **Write #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
outputlist	Optional. One or more comma-delimited numeric expressions or string expressions to write to a file.

Remarks

Data written with **Write #** is usually read from a file with **Input #**.

If you omit *outputlist* and include a comma after *filenumber*, a blank line is printed to the file. Multiple expressions can be separated with a space, a semicolon, or a comma. A space has the same effect as a semicolon.

When **Write #** is used to write data to a file, several universal assumptions are followed so the data can always be read and correctly interpreted using **Input #**, regardless of locale:

Numeric data is always written using the period as the decimal separator.

For Boolean data, either `#TRUE#` or `#FALSE#` is printed. The **True** and **False** keywords are not translated, regardless of locale.

Date data is written to the file using the universal date format. When either the date or the time component is missing or zero, only the part provided gets written to the file.

Nothing is written to the file if *outputlist* data is Empty. However, for Null data, `#NULL#` is written.

If *outputlist* data is **Null** data, `#NULL#` is written to the file.

For **Error** data, the output appears as `#ERROR errorcode#`. The **Error** keyword is not translated, regardless of locale.

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, that is, a carriage return–linefeed (**Chr(**13**)** + **Chr(**10**)**), after it has written the final character in *outputlist* to the file.

Note You should not write strings that contain embedded quotation marks, for example, `"1,2""X"` for use with the **Input #** statement: **Input #** parses this string as two complete and separate strings.

CallType Constants

The following constantscan be used anywhere in your code in place of the actual values:

Constant	Value	Decription
vbMethod	1	Indicates that a method has been invoked.
vbGet	2	Indicates a **Property Get** procedure.
vbLet	4	Indicates a **Property Let** procedure.
vbSet	8	Indicates a **Property Set** procedure.

The following constants can be used anywhere in your code in place of the actual values:

Constant	Value	Decription
vbModeless	0	**UserForm** is modeless.
vbModal	1	**UserForm** is modal (default).

Returns for CStr

If *expression* is	CStr returns
Boolean	A string containing **True** or **False**
Date	A string containing a date in the short date format of your system
Null	A run-time error
Empty	A zero-length string ("")
Error	A string containing the word **Error** followed by the error number
Other numeric	A string containing the number

Enum Statement

Declares a type for an enumeration.

Syntax

[**Public** | **Private**] **Enum** *name*

membername [= *constantexpression*]

membername [= *constantexpression*]

. . .

End Enum

The **Enum** statement has these parts:

Part	Description
Public	Optional. Specifies that the **Enum** type is visible throughout the project. **Enum** types are **Public** by default.
Private	Optional. Specifies that the **Enum** type is visible only within the module in which it appears.
name	Required. The name of the **Enum** type. The *name* must be a valid Visual Basic identifier and is specified as the

	type when declaring variables or parameters of the **Enum** type.
membername	Required. A valid Visual Basic identifier specifying the name by which a constituent element of the **Enum** type will be known.
constantexpression	Optional. Value of the element (evaluates to a **Long**). If no *constantexpression* is specified, the value assigned is either zero (if it is the first *membername*), or 1 greater than the value of the immediately preceding *membername*.

Remarks

Enumeration variables are variables declared with an **Enum** type. Both variables and parameters can be declared with an **Enum** type. The elements of the **Enum** type are initialized to constant values within the **Enum** statement. The assigned values can't be modified at run time and can include both positive and negative numbers. For example:

```
Enum SecurityLevel
    IllegalEntry = -1
    SecurityLevel1 = 0
    SecurityLevel2 = 1
End Enum
```

An **Enum** statement can appear only at module level. Once the **Enum** type is defined, it can be used to declare variables, parameters, or procedures returning its type. You can't qualify an **Enum** type name with a module name. **Public Enum** types in a class module are not members of the class; however, they are written to the type library. **Enum** types defined in standard modules aren’t written to type libraries. **Public Enum** types of the same name can't be defined in both standard modules and class modules, since they share the same name space. When two **Enum** types in different type libraries have the same name, but different elements, a reference to a variable of the type depends on which type library has higher priority in the **References**.

You can't use an **Enum** type as the target in a **With** block.

Event Statement

Declares a user-defined event.

Syntax

[**Public**] **Event** *procedurename* [**(***arglist***)**]

The **Event** statement has these parts:

Part	Description
Public	Optional. Specifies that the **Event** visible throughout the project. **Events** types are **Public** by default. Note that events can only be raised in the module in which they are declared.
procedurename	Required. Name of the event; follows standard variable naming conventions.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *varname*[**( )**] [**As** *type*]

Part	Description
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
varname	Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), Object, Variant, a user-defined type, or an object type.

Remarks

Once the event has been declared, use the **RaiseEvent** statement to fire the event. A syntax error occurs if an **Event** declaration appears in a standard module. An event can't be declared to return a value. A typical event might be declared and raised as shown in the following fragments:

```
' Declare an event at module level of a class m

Event LogonCompleted (UserName as String)

Sub
    RaiseEvent LogonCompleted("AntoineJan")
End Sub
```

Note You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, **Optional** arguments, or **ParamArray** arguments. Events do not have return values.