Caché Basic Reference

Version 2018.1
2019-09-20

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Caché Basic Reference
Caché Version 2018.1 2019-09-20

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book provides reference material for various elements of Caché Basic: commands, functions, constants, operators and symbols, and a list of the reserved words in Caché Basic.

This book contains the following sections:

- Symbols
- Caché Basic Commands
- Caché Basic Functions
- Caché Basic Operators
- Caché Basic Constants
- Caché Basic Objects
- Caché Basic General Concepts

There is also a detailed Table of Contents.

Other related topics in the Caché documentation set are:

- *Using Caché Basic*

For general information, see *Using InterSystems Documentation*.

Symbols

Symbols Used in Caché Basic

A table of characters used in Caché Basic as operators, etc.

Table of Symbols

The following are the literal symbols used in Caché Basic. (This list does not include symbols indicating format conventions, which are not part of the language.) There is a separate table for symbols used in ObjectScript.

The name of each symbol is followed by its ASCII decimal code value.

Symbol	Name and Usage
[space] or [tab]	*White space (Tab (9) or Space (32)):* One or more whitespace characters between keywords, identifiers, and variables.
"	*Quotes (34):* Used to enclose string literals. In Dynamic SQL used to enclose the SQL code as a string argument of the Prepare method.
""	*Double quotes:* Used to specify an empty string. Within a string literal, used to specify a literal double quote character.
%	*Percent sign (37):* Permitted first character in identifier names, such as variables, methods, and datatypes.
&	*Ampersand (38):* String concatenation operator. Numeric base prefix with Hex (&H) and Oct (&O) functions.
'	*Apostrophe (39):* Single-line comment indicator.
( )	*Parentheses (40,41):* Used to enclose a procedure or function parameter list. Used to nest expressions; nesting overrides the default order of operator precedence. Used to specify array subscripts. Enclose a test expression for an If, While, or in-line Case command.
*	*Asterisk (42):* Multiplication operator.
+	*Plus sign (43):* Addition operator.
,	*Comma (44):* Used to separate parameters in a procedure or function parameter list. Used to separate subscripts in an array. With Const and Dim commands, used to separate multiple assignments.
–	*Minus sign (45):* Unary arithmetic negative operator. Subtraction operator.
.	*Period (46):* Decimal point character. A valid character in global or process-private global names. Cannot be used in local variable names. Object dot syntax used to refer to a method or property of an object: `person.Name`.
/	*Slash (47):* Division operator (keep remainder).

Symbol	Name and Usage
:	*Colon (58):* Label suffix. For example, `LabelOne:`. Statement divider, used to separate multiple statements on the same line. For example, `Print a : Print b`. With Case function, used to associate case:value pairs.
<	*Less than (60):* Less than operator.
<=	*Less than or equal to:* Less than or equal to operator.
<>	*Less than/Greater than:* Inequality operator.
=	*Equal sign (61):* Equals comparison operator. Assignment operator.
>	*Greater than (62):* Greater than operator.
>=	*Greater than or equal to:* Greater than or equal to operator.
?	*Question mark (63):* In Dynamic SQL, an input parameter variable supplied by the Execute method.
@	*At sign (64):* Calling function syntax: func@Routine(args).
E, e	*The letter "E" (69, 101):* Base-10 exponent (scientific notation) numeric literal.
\	*Backslash (92):* Integer division operator (drop remainder).
^	*Caret (94):* Global variable name prefix; for example, ^myglobal. Exponentiation operator.
^|	*Caret bar:* depending on the character(s) that follow, this may be either: An extended global reference, a global reference where a pair of bars encloses a null string, or a quoted namespace or directory name. The bars and their contents are not part of the global name. For example: ^|""|globname, or ^|"namespace"|globname. A process-private global with the prefix ^||. The bars *are* part of the process-private global name. For example, ^||ppgname. Also valid as syntax for this process-private global: ^|"^"|ppgname.
_	*Underscore (95):* Line continuation character. A line ending with an underscore continues on next line. Commonly preceded by one or more blank spaces. A valid character in local variable names or routine names. Cannot be used in global variable or process-private global variable names.
{ }	*Curly braces (123,125):* Code block delimiters used in procedures.

Caché Basic Commands

Call

Transfers control to a **Sub** procedure or **Function** procedure.

```
[Call] name([arglist])
```

Arguments

The **Call** statement syntax has these parts:

Call	**Call** is an optional keyword. If specified, you must enclose *arglist* in parentheses. For example: `Call MyProc(0)`
name	Name of the procedure to call. To call a procedure in an external routine, specify *name@routine(arglist)*.
arglist	*Optional* — Comma-delimited list of variables, arrays, or expressions to pass to the procedure. The parentheses are required, even when there are no arguments.

Description

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 *arglist*. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's return value is discarded.

To omit an *arglist* argument value, you must specify an undefined variable. This is a significant difference between ObjectScript and Caché Basic. In ObjectScript an omitted argument can be specified using a placeholder comma. In Caché Basic you cannot use a placeholder comma; you must supply an undefined named variable.

Examples

The following example shows how to use the **Call** statement:

```
Call MyFunction("Hello World")

Function MyFunction(text)
  Println text
End Function
```

Catch

Identifies a block of code to execute when an exception occurs.

```
Try
   statements
Catch [exceptionvar]
   statements
End Try
```

Arguments

exceptionvar	*Optional* — An exception variable. Specified as a local variable, with or without subscripts, that receives a reference to a Caché Object.

Description

The **Catch** command defines an exception handler, one or more statements to execute when an exception occurs in the code following a **Try** statement. The **Catch** command is followed by one or more exception handling code statements. The **Catch** block must immediately follow its **Try**, and the paired **Try** and **Catch** are terminated by an **End Try** statement.

The **Catch** command has two forms:

- Without an argument
- With an argument

Catch without an Argument

Argumentless **Catch** execute the series of statements between **Catch** and **End Try**.

Catch with an Argument

Catch *exceptionvar* receives a Caché Object reference (oref) from the **Throw** command or from the system runtime environment in the event of a system error. This Object provides properties that contain information about the exception, such as the Name of the error and the Location where it occurred. The user-written **Catch** exception handler code can use this information to analyze the exception.

Arguments

exceptionvar

A local variable, used to receive the exception object reference from the **Throw** command or from the system runtime environment in the event of a system error. When a system error occurs, *exceptionvar* receives a reference to an object of type %Exception.SystemException. For further details, refer to the %Exception.AbstractException class in the *InterSystems Class Reference*.

Examples

The following example shows a **Catch** invoked by a runtime error. The *myvar* argument receives a system-generated exception object:

```
Try
  PRINTLN "about to divide by zero"
  SET a=7/0
  PRINTLN "this should not display"
Catch myvar
  PRINTLN "this is the exception handler"
  PRINTLN "Error is: ",Err.Description
  PRINTLN "Error code: ",myvar.Code
End Try
PRINTLN "this is where the code falls through"
```

See Also

- Throw command
- Try command
- Err object

Const

Declares constants for use in place of literal values.

```
Const constname = expression
```

Arguments

The **Const** statement syntax has these parts:

constname	Name of the constant; follows standard variable-naming conventions.
expression	Literal or any combination that includes all arithmetic or logical operators except **Is**.

Description

To combine several constant declarations on the same line, separate each constant assignment with a comma.

You cannot use variables, user-defined functions, or intrinsic Caché Basic functions (such as **Chr**) in constant declarations. By definition, they cannot be constants. Constants declared in a **Sub** or **Function** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the script in which it is declared. You can use constants anywhere you can use an expression.

Examples

The following code illustrates the use of the **Const** statement:

```
Const MyVar = 459

' Declare multiple constants on same line.
Const MyStr = "Hello", MyNumber  = 3.4567
```

Notes

Constants can make your scripts self-documenting and easy to modify. Unlike variables, constants cannot be inadvertently changed while your script is running.

See Also

- Dim Statement
- Function Statement
- Sub Statement

Continue

Jumps to FOR or DO WHILE statements and reexecutes test and loop.

```
Continue Do
Continue For
```

Arguments

The **Continue Do** and **Continue For** statements do not have any arguments

Description

The **Continue Do** or **Continue For** statement is used within the code block following a **For** or **Do While** statement. **Continue Do** or **Continue For** causes execution to jump back to the **For** or **Do While** statement and to evaluate its test condition, and, based on that evaluation, reexecutes the code block loop.

Examples

The following example illustrates the use of the **Continue** statement:

```
For i = 1 to 10
  If i = 5 Then
  Continue For
  Println i
  End If
Next
```

See Also

- Do...Loop Statement
- Exit Statement
- For Each...Next Statement
- For...Next Statement

Copy

Copies array elements from source to target.

```
Copy target=source
```

Arguments

The **Copy** statement has the following parameters:

source	The name of the variable, typically an array, which should be copied.
target	The name of the variable into which the contents of source should be copied.

Description

All nodes in the target variable are deleted prior to the copy process. The only difference between **Copy** and **Merge** is the deletion of the target nodes.

Examples

```
Erase source, target

target(1) = "node 1"
target(1,1) = "node 1,1"
target(2) = "node 2"
target(3,1) = "node 3,1"
source(3,2) = "node 3,2"

Copy target = source

Println Exists(target(3,1)) 'not defined anymore, returns 0
Println Exists(target(3,2)) 'does now exist, returns 1
Println Exists(target(1))   'not defined and has no subnodes, returns 0
```

See Also

- Merge Statement

Debug

Interrupts program execution and enters programmer mode.

```
Debug
```

Arguments

None.

Description

The **Debug** statement interrupts execution of the current routine and returns control to programmer mode. Once in programmer mode, you can perform debugging operations. A **Debug** statement included in code sets a breakpoint, which interrupts routine execution and returns the process to programmer mode.

The **Debug** statement is functionally equivalent to the ObjectScript argumentless BREAK command.

Dim

Declares variables.

```
Dim varname[, varname] . . .
```

Arguments

The **Dim** statement syntax has these parts:

varname	Name of the variable; follows standard variable naming conventions.

Description

Variables declared with **Dim** at the script level are available to all procedures within the script. At the procedure level, variables are available only within the procedure.

All uninitialized variables are treated as zero-length strings ("").

Examples

The following examples illustrate the use of the Dim statement:

```
Dim MyStr        ' Declare one variable
Dim MyVar, MyNum ' Declare two variables
```

Notes

Caché Basic does not require the dimension of arrays to be specified, and therefore does not implement the **ReDim** Statement.

See Also

- Set Statement

Do...Loop

Repeats a block of statements while a condition is True or until a condition becomes True.

```
Do [{While | Until} condition]
    [statements]
    [Exit Do]
    [statements]
Loop

Or, you can use this syntax:

Do
    [statements]
    [Exit Do]
    [statements]
Loop [{While | Until} condition]
```

Arguments

The **Do...Loop** statement syntax has these parts:

condition	Numeric or string expression that is **True** or **False**.
statements	One or more statements that are repeated while or until condition is **True**.

Description

The **Exit Do** can only be used within a **Do...Loop** control structure to provide an alternate way to exit a **Do...Loop**. Any number of **Exit Do** statements may be placed anywhere in the **Do...Loop**. Often used with the evaluation of some condition (for example, **If...Then**), **Exit Do** transfers control to the statement immediately following the **Loop**.

When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is nested one level above the loop where it occurs.

Examples

The following examples illustrate use of the **Do...Loop** statement:

```
Do Until MyNum = 6
  MyNum = Int (6 * Rnd + 1)  ' Generate a random integer between 1 and 6
  Println MyNum
Loop

Dim Check, Counter
Check = True: Counter = 0   ' Initialize variables.
Do                          ' Outer loop.
  Do While Counter < 20     ' Inner loop.
    Counter = Counter + 1   ' Increment Counter.
    If Counter = 10 Then    ' If condition is True...
      Check = False         ' set value of flag to False.
      Exit Do               ' Exit inner loop.
    End If
  Loop
Loop Until Check = False    ' Exit outer loop immediately.
```

See Also

- Exit Statement
- For...Next Statement
- While...Wend Statement

Erase

Removes the named variable and deallocates dynamic-array storage space.

```
Erase varname
```

Arguments

The **Erase** statement has the following argument:

varname	The name of the variable to be erased.

Description

The **Erase** statement removes the variable and all descended nodes.

Erase may be used to insure that a variable has no defined value, such as when a named variable is used as a placeholder in an argument list.

Examples

The following example uses **Erase** to remove an array and its subnodes:

```
array = "root node"
array("subnode") = "subnode"
array("subnode", "subnode") = "subnode, subnode"
Println Exists(array) 'returns 3; variable defined and has array elements
Erase array
Println Exists(array) 'returns 0
```

The following example uses **Erase** to specify an explicitly undefined placeholder variable:

```
  Erase blankvar
  tStatement = New %SQL.Statement(blankvar,"Sample")
  PrintLn "Success"
```

See Also

- EraseArray Statement
- EraseValue Statement

EraseArray

Removes the array elements of a variable and deallocates dynamic-array storage space.

```
EraseArray varname
```

Arguments

The **EraseArray** statement has the following argument:

varname	The name of the variable for which the array elements should be erased.

Description

The **EraseArray** statement removes array elements of the variable, but not the root node.

Examples

The following example demonstrates the use of the **EraseArray** statement:

```
array = "root node"
array("subnode") = "subnode"
array("subnode", "subnode") = "subnode, subnode"
Println Exists(array) 'returns 3; variable defined and has array elements
EraseArray array
Println Exists(array) 'returns 1
```

See Also

- Erase Statement
- EraseValue Statement

EraseValue

Removes the root node of a variable.

```
EraseValue varname
```

Arguments

The **EraseValue** statement has the following argument:

varname	The name of the variable for which the root node should be erased.

Description

The **EraseValue** statement removes the root nodes of the variable, but does not delete the array elements.

Examples

The following example demonstrates the use of the **EraseArray** statement:

```
array = "root node"
array("subnode") = "subnode"
array("subnode", "subnode") = "subnode, subnode"
Println Exists(array) 'returns 3; variable defined and has array elements
EraseValue array
Println Exists(array) 'returns 2
```

See Also

- Erase Statement
- EraseArray Statement

Exit

Exits a block of **Do...Loop**, **For...Next**, **Function**, or **Sub** code.

```
Exit Do
Exit For
Exit Function
Exit Sub
```

Arguments

The Exit statement syntax has these forms:

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 it 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 it 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 Sub	Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub**.

Examples

The following example illustrates the use of the **Exit** statement:

```
Sub RandomLoop
  Dim I, MyNum
  Do                              ' Set up infinite loop.
    For I = 1 To 1000             ' Loop 1000 times.
      MyNum = Int(Rnd * 100)      ' Generate random numbers.
      Select Case MyNum           ' Evaluate random number.
        Case 17: Print "Case 17"
          Exit For                ' If 17, exit For...Next.
        Case 29: Print "Case 29"
          Exit Do                 ' If 29, exit Do...Loop.
        Case 54: Print "Case 54"
          Exit Sub                ' If 54, exit Sub procedure.
        End Select
    Next
  Loop
End Sub
```

See Also

- Continue Statement
- Do...Loop Statement
- For Each...Next Statement

- For...Next Statement
- Function Statement
- Sub Statement

For Each...Next

Repeats a group of statements for each element in an array or collection.

```
For Each element In group
    [statements]
    [Exit For]
    [statements]
Next [element]
```

Arguments

The **For Each...Next** statement syntax has these parts:

element	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	Name of an object collection or array.
statement	One or more statements that are executed on each item in group.

Description

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. As long as 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.

The **Exit For** can only be used within a **For Each...Next** or **For...Next** control structure to provide an alternate way to exit. Any number of **Exit For** statements may be placed anywhere in the loop. The **Exit For** is often used with the evaluation of 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.

For Each and the Split Function

A Split function cannot be directly used as the *group* argument of a **For Each...Next** statement. You must first assign the **Split** return value to an array variable. You can then specify this array variable as the *group* argument of the **For Each...Next** statement.

Examples

The following example illustrates use of the **For Each...Next** statement:

```
Erase ^RandomData

' Generate some random nodes
For i =  65 to 90
  If Rnd(i) > .5 Then
    ^RandomData(Chr(i),"subnode")="data"
  Else
    ^RandomData(Chr(i))="data"
  End If
Next

PrintLn "Traverse forwards"

For each k1 in ^RandomData
  PrintLn k1
  For each k2 in ^RandomData(k1)
    Print k1,vbTAB,k2
    If Exists(^RandomData(k1,k2)) and vbHasValue Then
```

```
      Print " = ",^RandomData(k1,k2)
    End If
    PrintLn
  Next
Next
```

Notes

If you omit element in a **Next** statement, execution continues as if you had included it. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

See Also

- Do...Loop Statement
- Exit Statement
- For...Next Statement
- While...Wend Statement

For...Next

Repeats a group of statements a specified number of times.

```
For counter = start To end [Step step]
    [statements]
    [Exit For]
    [statements]
Next
```

Arguments

The **For...Next** statement syntax has these parts:

counter	Numeric variable used as a loop counter. The variable cannot be an array element or an element of a user-defined type.
start	Initial value of counter.
end	Final value of counter.
step	Amount counter is changed each time through the loop. If not specified, step defaults to one.
statements	One or more statements between For and Next that are executed the specified number of times.

Description

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

Once the loop starts and 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.

Exit For can only be used within a For Each...Next or For...Next control structure to provide an alternate way to exit. Any number of Exit For statements may be placed anywhere in the loop. Exit For is often used with the evaluation 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
          ' Some statements
        Next
    Next
Next
```

Notes

Changing the value of counter while inside a loop can make it more difficult to read and debug your code.

See Also

- Do...Loop Statement
- Exit Statement
- For Each...Next Statement
- While...Wend Statement

Function

Declares the name, arguments, and code that form the body of a Function procedure.

```
[Public | Private] Function name [(arglist)] [ As classname ]
    [statements]
    [name = expression]
    [Exit Function]
    [statements]
    [name = expression]
End Function
```

Arguments

The **Function** statement syntax has these parts:

Public	*Optional* — Keyword indicating that the **Function** procedure is accessible to all other procedures in all scripts.
Private	*Optional* — Keyword indicating that the **Function** procedure is accessible only to other procedures in the script where it is declared.
name	Name of the **Function**. Follows local variable naming conventions.
arglist	*Optional* — List of variables representing arguments that are passed to the **Function** procedure when it is called, separated by commas.
classname	*Optional* — Name of the class of the return value.
statements	Any group of statements to be executed within the body of the **Function** procedure.
expression	*Optional* — Return value of the **Function**.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] varname[( )]

ByVal	Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference.
varname	Name of the variable representing the argument; follows standard variable naming conventions.

Description

Function procedures are visible to all other procedures in your script. The value of local variables in a **Function** is not preserved between calls to the procedure.

All executable code must be contained in the procedure. Nesting is not permitted; you cannot define a **Function** procedure inside another **Function** or **Sub** 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.

There are two ways to return a value from a **function**: you can specify the value on a **Return** statement, or you can 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 zero-length string (""). A function that returns an object reference returns a zero-length string ("") if no object reference is assigned to name within the **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.

All variables in a Caché Basic **Function** procedure are private. Therefore, a **Function** procedure cannot access public variables, such as SQLCODE. To use public variables, use a top-level Caché Basic routine, rather than a called function or subroutine.

To omit an *arglist* argument value, you must specify an undefined variable. This is a significant difference between ObjectScript and Caché Basic. In ObjectScript an omitted argument can be specified using a placeholder comma. In Caché Basic you cannot use a placeholder comma; you must supply an undefined named variable.

Examples

The following example shows both ways to assign a return value. First by specifying “True” to the Return statement, then by assigning “False” to the function named IsGreaterThan. False is assigned to the function name to indicate that an invalid value was found.

```
Function IsGreaterThan(lower, upper)
If lower < upper Then Return True
IsGreaterThan = False
End Function
```

Notes

Function procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

See Also

- Call Statement
- Dim Statement
- Return Statement
- Sub Statement

Goto

Transfers program execution to the specified location.

```
Goto label
```

Arguments

label	A line label specifying the target of the **Goto** operation. A label is a valid identifier, followed by a colon suffix. See Labels in *Using Caché Basic*. The **Goto** *label* reference can be specified with or without a colon suffix.

Description

The **Goto** statement immediately shifts program execution to the line location in the program specified by the *label*. The specified line must be in the same procedure as the **Goto** statement, or a compile-time error occurs.

The *label* argument specifies an existing label in the current program. Specifying the label's colon suffix is optional. Label names are case-sensitive. Specifying a non-existent label name results in a runtime error.

Examples

The following example illustrates the use of the **Goto** statement. Note that the *label* argument can include or omit the colon suffix:

```
Mod1:
  Println "Mod1"
  Goto Mod2
  Println "skipped over"
Mod2:
  Println "Mod2"
  Goto Mod4:
Mod3:
  Println "skipped Mod3"
Mod4:
  Println "Mod4"
```

The following example illustrates that more than one label can appear on a single line:

```
Mod1:
  Println "Mod1"
  Goto Mod3:
  Println "skipped over"
Mod2: Mod3:
  Println "Mods 2 and 3"
  Goto Mod4:
Mod4:
  Println "Mod4"
```

See Also

- Basic: On Error Goto statement
- ObjectScript: GOTO command
- Labels in the “Lexical Structure” chapter of *Using Caché Basic*.

If...Then...Else

Conditionally executes a group of statements, depending on the value of an expression.

```
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
```

Arguments

The **If...Then...Else** statement syntax has these parts:

condition	An expression that evaluates to **True** or **False**.
statements	One or more statements separated by colons; executed if condition is **True**.
condition-n	Same as condition.
elseifstatements	One or more statements executed if the associated condition-n is **True**.
elsestatements	One or more statements executed if no previous condition or condition-n expression is True.

Description

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.

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** (if any) is evaluated in turn. When a **True** condition is found, the statements following the associated **Then** are executed. If none of the **ElseIf** statements are **True** (or 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**.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** statements as you want in a block **If**, but none can appear after the **Else** clause. Block **If** statements can be nested; that is, contained within one another.

What follows the **Then** keyword is examined to determine whether or not a statement is a block **If**. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

A block **If** statement must be the first statement on a line. The block **If** must end with an **End If** statement.

Notes

With the single-line syntax, it is possible to have multiple statements executed as the result of an **If...Then** decision, but they must all be on the same line and separated by colons, as in the following statement:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

See Also

- Case Function

Imports

Imports a package name.

```
Imports package [,package2 [,...]]
```

Arguments

The **Imports** statement syntax has these parts:

package	A package name, or a comma-separated list of package names.

Description

You use the **Imports** statement to import a package. This allows statements to append a package name to a class without having to explicitly declare the package name each time. If the package does not exist, or if the specified class is not found in any of the imported packages, or if the specified class is found in more than one imported package, no package name is appended to the class name.

Examples

The following example illustrates the use of the **Imports** statement:

```
Imports %Library
MyObject = new MsgHandler
```

which is equivalent to:

```
MyObject = new %Library.MsgHandler
```

See Also

- System Object

Input

Accepts input and stores it in a variable.

```
Input data
```

Arguments

data	Either the name of a variable used to receive the data input, or a quoted string specifying the data.

Description

The **Input** statement inputs a literal data value. It can interactively receives a data value from the user into a variable, or it can input a specified quoted string.

Input with a variable cannot be used in a program running in background. Program execution is paused until the user indicates the end of data input and submits the data value by pressing the Return key.

Input does not time out.

The ObjectScript **READ** command provides more extensive support for interactive user input.

Examples

The following example illustrates the interactive use of the **Input** statement:

```
Println "Type your name, then press Return"
Input namevar
Println "Thanks ",namevar
```

The following example illustrates the background use of the **Input** statement:

```
Println "Here's the name"
Input "Fred"
Println
Println "Thanks "
```

See Also

- Basic: Print statement
- ObjectScript: READ command

Let

Assigns an object reference to a variable or property.

```
Let objectvar = objectexpression
```

Arguments

objectvar	Name of the variable or property; follows standard variable-naming conventions.
objectexpression	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.

Description

In Caché Basic, the **Let** statement is functionally identical to the **Set** statement. Refer to the Set statement for further details.

Merge

Merges array elements from source to target.

```
Merge target=source
```

Arguments

The **Merge** statement has the following parameters:

source	The name of the variable, typically an array, which should be merged into the target variable.
target	The name of the variable into which the contents of source should be merged.

Description

Nodes in the source variable overwrite corresponding nodes in the target variable, and all descendents of source overwrite corresponding descendents of target. All other target nodes are unchanged. The only difference between **Copy** and **Merge** is that **Copy** deletes the target nodes, and **Merge** does not.

Examples

```
Erase source, target

target(1) = "node 1"
target(1,1) = "node 1,1"
target(2) = "node 2"
target(3,1) = "node 3,1"
source(3,2) = "node 3,2"

Merge target=source

Println Exists(target(3,1)) 'is still defined, returns 1
Println Exists(target(3,2)) 'does now exist, returns 1
```

See Also

- Copy Statement

New

Creates a new instance of an object.

```
New object
```

Arguments

The **New** statement has the following argument:

object	Name of the object for which a new instance should be created.

Examples

The following examples demonstrate how to use the **New** statement:

```
person = New User.Person
output = New %File("\PersonList.txt")
```

See Also

- IsObject Function
- OpenId Statement

On Error Goto

Enables an error-handling routine and specifies the location of the routine within a procedure.

```
On Error GoTo [ label | 0 ]
```

Arguments

The **On Error GoTo** statement has the following argument:

label	A line label specifying the target of the **Goto** operation. A label is a valid identifier, followed by a colon suffix. See Labels in *Using Caché Basic*.

Description

Enables the error-handling routine that starts at the line specified by the *label* argument. If a runtime error occurs, control branches to the specified line, making the error handler active. The specified line must be in the same procedure as the **On Error** statement, or a compile-time error will occur.

The *label* argument specifies an existing label in the current program. Specifying the label's colon suffix is optional. Label names are case-sensitive. Specifying a non-existent label name results in a runtime error.

Use **On Error Goto 0** to disable error handling if you have previously enabled it.

When **On Error Goto** is triggered by an error, it is automatically disabled. This means that the occurrence of a second error causes a program abort, rather than initiating an infinite loop.

An error-handling routine is not a **Sub** procedure or a **Function** procedure. It is a section of code marked by a line label.

Error-handling routines rely on the value in the Number property of the **Err** object to determine the cause of the error. The 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**.

Examples

The following example shows the use of the **On Error Goto** statement. Here the error is attempting to divide 6 by 0. The ErrMod error handler displays the error number (18) and description:

```
Mod1:
  On Error Goto ErrMod
  PrintLn "Mod1 pre-div"
  Println "result: ",6/0
  Println "Mod1 post-div"
  Goto Done
ErrMod:
  Println "Handling an error!"
  PrintLn "Error ",Err.Number," ",Err.Description
Done:
  Println "All done"
```

In the following example, the ErrMod error handler corrects the division by zero problem by changing *divisor* to 1, and retries the Mod1 operation. Note that invoking the ErrMod error handling module resets **On Error Goto**, so that the occurrence of the second error in this program (attempting to divide 5 by 0) aborts the program, rather than calling ErrMod again:

```
Setup:
  On Error Goto ErrMod
  divisor=0
Mod1:
  PrintLn "Mod1 pre-div"
  Println "result: ",6/divisor
  Println "Mod1 post-div"
  Println 5/0
  Goto Done
ErrMod:
  Println "Handling an error!"
  PrintLn "Error ",Err.Number," ",Err.Description
  If Err.Number=18 Then
     divisor=1
     Goto Mod1
  Else
     Println Err.Number
  End If
Done:
  Println "All done"
```

The following example shows the use of the **On Error GoTo** statement in a user-defined function:

```
PrintLn ErrorTest(1)
PrintLn ErrorTest(0)

Function ErrorTest(Arg)
  On Error Goto ErrDisplay
  return 1/Arg
ErrDisplay:
  PrintLn "Error ", Err.Number, " ", Err.Description, " ", Err.Source
  Err.Clear
  Return 0
End Function
```

See Also

- Goto statement
- Err object
- Labels in the "Lexical Structure" chapter of *Using Caché Basic*.

OpenId

Opens a new instance of an object for a given Identifier.

```
OpenId object
```

Arguments

The **OpenId** statement has the following argument:

object	Name of the object for which a new instance should be created.

Examples

The following example demonstrate how to use the **OpenId** statement:

```
person = OpenId User.Person(5012)
  'Instantiates a person object with the Id of 5012
```

See Also

- IsObject Function
- New Statement

Option Explicit

Used at script level to force explicit declaration of all variables in that script.

```
Option Explicit
```

Arguments

none

Description

If used, the **Option Explicit** statement must appear in a script before any procedures.

When you use the **Option Explicit** statement, you must explicitly declare all variables using the **Dim** statement. If you attempt to use an undeclared variable name, an error occurs.

Note: You cannot use **Option Explicit** inside a method.

Examples

The following example illustrates use of the **Option Explicit** statement:

```
Option Explicit    ' Force explicit variable declaration.
Dim MyVar          ' Declare variable.
MyInt = 10         ' Undeclared variable generates error.
MyVar = 10         ' Declared variable does not generate error.
```

Notes

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.

Print, Println

Writes a string to the current device.

```
Print expr
Println expr
```

Arguments

expr	An expression that is evaluated and written to the current device. This can be a single expression, or a comma-separated list of expressions.

Description

The **Print** statement is used to write an expression (or a list of expressions) to the current device. The **Println** statement is identical to **Print**, except that it automatically appends vbCRLF (carriage return / line feed) after writing the last expression in the list.

Examples

The following example demonstrate the use of the **Print** and **Println** commands with strings and string variables. To include a quote character within a string, double the quote character. Printing the empty string ("") completes without error and can be used with **Println** to insert a blank line. An undefined variable (*z* in this example) is treated the same as the empty string. Note that variable names are case-sensitive.

```
Set a="big "
Set b="bad "
Set c="bug"
Print "Hello"
Println " world!"
Println ""
Println "this is a quote ("") character"
Println z
Println a,b
Println c
Print a,b
Print c
```

The following example demonstrate the use of the **Print** and **Println** commands with numeric expressions. Caché converts numbers to canonical form, removing unnecessary signs and leading and trailing blanks. It then evaluates arithmetic expressions. Numbers specified as string are passed as literals without conversion.

```
Set x="++007.9900"
Set y=++007.9900
Println 123456
Println (3+3)*2
Println 3+(3*2)
Println +007.9900
Println x
Println y
```

The following example demonstrate the use of the **Print** and **Println** commands with subscripted global variables:

```
Set ^a(1)="fruit"
Set ^a(1,1)="apple"
Println "An ",^a(1,1)," is a ",^a(1)
```

See Also

- Basic: Set command
- ObjectScript: WRITE command

Randomize

Initializes the random-number generator.

```
Randomize [number]
```

Arguments

The number argument can be any valid numeric expression.

Description

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.

Examples

The following example illustrates use of the **Randomize** statement:

```
Dim MyValue, Response
Randomize    ' Initialize random-number generator.

MyValue = Int((6 * Rnd) + 1)    ' Generate random value between 1 and 6.

Println MyValue
```

Notes

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.

See Also

- Rnd Function

Rem

Used to include explanatory remarks in a program.

```
Rem comment
or

' comment
```

Arguments

None.

The comment argument is the text of any comment you want to include. After the **Rem** keyword, a space is required before comment.

Description

As shown in the syntax section, you can use an apostrophe (') instead of the **Rem** keyword. If the **Rem** keyword follows other statements on a line, it must be separated from the statements by a colon. However, when you use an apostrophe, the colon is not required after other statements.

Examples

The following example illustrates the use of the **Rem** statement:

```
Dim MyStr1, MyStr2
MyStr1 = "Hello" : Rem Comment after a statement separated by a colon.
MyStr2 = "Goodbye"     ' This is also a comment; no colon is needed.
Rem Comment on a line with no code; no colon is needed.
```

Return

Exits from the current function and returns a value from that function.

```
Return expression
```

Arguments

The **Return** statement syntax has these parts:

expression	Any numeric or string expression.

Description

You use the **Return** statement to stop execution of a function and return the value of expression. If no **Return** statement is executed from within the function, the expression that called the current function is assigned the value undefined.

Examples

The following example illustrates the use of the **Return** statement:

```
Function IsGreaterThan(lower, upper)
If lower > upper Then
    Return False
ElseIf lower = upper Then
    Return False
Else
    Return True
End If
End Function
```

See Also

- Function Statement

Select Case

Executes one of several groups of statements, depending on the value of an expression.

```
Select Case testexpression
    [Case expressionlist-n
        [statements-n]] . . .
    [Case Else elsestatements]
End Select
```

Arguments

The **Select Case** statement syntax has these parts:

testexpression	Any numeric or string expression.
expressionlist-n	Required if Case appears. Delimited list of one or more expressions.
statements-n	One or more statements executed if *testexpression* matches any part of *expressionlist-n*.
elsestatements	One or more statements executed if *testexpression* does not match any of the Case clauses. There can be only one Case Else clause with only one set of *elsestatements*.

Description

If testexpression matches (is equal to) 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**.

Select **Case** statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

Examples

The following example illustrates the use of the **Select Case** statement:

```
Dim Color, MyVar
Sub ChangeBackground (Color)
  MyVar = lcase (Color)
     Select Case MyVar
           Case "red"    document.bgColor = "red"
           Case "green"  document.bgColor = "green"
           Case "blue","azure"   document.bgColor = "blue"
           Case Else     Print "pick another color"
     End Select
End Sub
```

See Also

- If...Then...Else Statement

Set

Assigns an object reference to a variable or property.

```
Set objectvar = objectexpression
```

Arguments

objectvar	Name of the variable or property; follows standard variable-naming conventions.
objectexpression	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.

Description

The **Set** statement is used to assign a value to a variable. This value can be a number, a string, or an object reference. The variable can be a local variable, a process-private global variable, or a global variable, and can be subscripted. For further details on types of variables, see Identifiers and Variables in *Using Caché Basic*.

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 these variables are references to (rather than copies of) the object, any change in the object is reflected in all variables that refer to it.

Examples

The following example shows two **Set** statements assigning a string to a variable. The second **Set** statement uses the concatenation operator to construct the string:

```
Set a = "the quick brown fox"
Println a
Set b = "the "&"quick "&"brown "&"fox"
Println b
```

The following example shows two **Set** statements that assign a string to a subscripted global variable:

```
Set ^a(1)="fruit"
Set ^a(1,1)="apple"
Println "An ",^a(1,1)," is a ",^a(1)
```

The following example shows how to assign an object reference to a variable:

```
Set person = New User.Person()
Println person
```

See Also

- Basic: Dim Statement
- ObjectScript: SET command

Sleep

Causes program execution to delay for the specified number of seconds.

```
Sleep time
```

Arguments

time	A number specifying the number of seconds to delay. Can be an integer specifying whole seconds, or a decimal number specifying fractional seconds.

Description

The **Sleep** statement delays program execution for the specified duration in seconds. Program execution then resumes at the statement immediately following **Sleep**. A non-numeric *time* value is treated as zero.

Examples

The following example illustrates the use of the **Sleep** statement with whole seconds:

```
Println now
Sleep 5
Println now
```

The following example illustrates the use of the **Sleep** statement with fractional seconds:

```
Println now
Sleep 0.9
Println now
```

See Also

- Basic: Now function, Timer function
- ObjectScript: HANG command

Sub

Declares the name, arguments, and code that form the body of a **Sub** procedure.

```
[Public | Private]Sub name [(arglist)]
    [statements]
    [Exit Sub]
    [statements]
End Sub
```

Arguments

The **Sub** statement syntax has these parts:

Public	*Optional* — Keyword indicating that the **Sub** procedure is accessible to all other procedures in all scripts.
Private	*Optional* — Keyword indicating that the **Sub** procedure is accessible only to other procedures in the script where it is declared.
name	Name of the **Sub**. Follows local 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	Any group of statements to be executed within the body of the **Sub** procedure.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] varname[( )]

ByVal	Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference.
varname	Name of the variable representing the argument; follows standard variable naming conventions.

Description

The value of local variables in a **Sub** procedure is not preserved between calls to the procedure.

All executable code must be contained in the procedure. Nesting is not permitted; you cannot define a **Sub** procedure inside another **Sub** or **Function** procedure.

The **Exit Sub** statement causes an immediate exit from a **Sub** procedure. Program execution continues with the statement following the statement that called the **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in a **Sub** procedure.

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 cannot 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**) 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.

All variables in a Caché Basic **Sub** procedure are private. Therefore, a **Sub** procedure cannot access public variables, such as SQLCODE. To use public variables, use a top-level Caché Basic routine, rather than a called subroutine.

To omit an *arglist* argument value, you must specify an undefined variable. This is a significant difference between ObjectScript and Caché Basic. In ObjectScript an omitted argument can be specified using a placeholder comma. In Caché Basic you cannot use a placeholder comma; you must supply an undefined named variable.

Notes

Sub procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script 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 script-level name. Explicitly declare variables to avoid this kind of conflict. You can use an **Option Explicit** statement to force explicit declaration of variables.

See Also

- Call Statement
- Dim Statement
- Function Statement

TCommit

Marks the successful completion of a transaction.

```
TCommit
```

Arguments

The **TCommit** statement does not have any arguments.

Description

TCommit marks the successful end of a transaction initiated by the corresponding **TStart**.

Examples

The following example illustrates the use of the **TCommit** statement:

```
TStart
If StorePerson(personobject) Then
 TCommit
Else
 TRollback
End If
```

See Also

- TRollback Statement
- TStart Statement

Throw

Throws an exception from a Try block to a Catch exception handler.

```
Throw [oref]
```

Arguments

oref	*Optional* — A user-defined object reference.

Description

The **Throw** command explicitly issues an exception from within a block of code defined by a **Try** command. Issuing a **Throw** transfers execution from the **Try** block to the corresponding **Catch** exception handler.

Throw is used to issue an explicit exception. Caché issues an implicit exception when a runtime error occurs. A runtime error generates an exception object which it throws to a **Catch** exception handler.

Throw has two forms:

- Without an argument
- With an argument

Throw without an Argument

Argumentless **Throw** transfers error processing to the corresponding **Catch** error handler. No object is pushed on the stack, but the %New() method is called.

Throw with an Argument

Throw *oref* specifies a user-defined object reference, which it throws to the **Catch** command, pushing it on the execution stack. The calling of the %New() method is optional.

Arguments

expression

A user-defined object reference (oref). For example, `Throw ##class(%Exception).%New()`. The creation and population of this exception object is the responsibility of the programmer.

Examples

The following example shows an argumentless **Throw**:

```
Try
  SET x=2
  PRINTLN "about to divide by ",x
Throw
  SET a=7/x
  PRINTLN "Success: the result is ",a
Catch myvar
  PRINTLN "this is the exception handler"
  PRINTLN "Error number: ",Err.Number
  PRINTLN "Error is: ",Err.Description
  PRINTLN "Error code: ",myvar.Code
END Try
PRINTLN "this is where the code falls through"
```

See Also

- Catch command

- Try command
- Err object

TRollback

Rolls back (reverts) changes made during the current transaction.

```
TRollback
```

Arguments

The **TRollback** statement does not have any arguments.

Description

TRollback terminates the current transaction and restores all journaled database values to the values they held at the start of the transaction.

Caché Basic does not support nested transactions. A **TRollback** returns the transaction level ($TLEVEL) to 0, regardless of how many nested **TStart** statements have been issued.

Examples

The following example illustrates the use of the **TRollback** statement:

```
TStart
If StorePerson(personobject) Then
 TCommit
Else
 TRollback
End If
```

See Also

- TCommit Statement
- TStart Statement

Try

Identifies a block of code to monitor for errors during execution.

```
Try
   statements
Catch [exceptionvar]
   statements
End Try
```

Description

The **Try** command takes no arguments. It is used to identify one or more Caché Basic code statements between the **Try** keyword and the **Catch** keyword. This block of code is protected code for structured exception handling. If an exception occurs within this block of code, Caché sets **Err**, then transfers execution to an exception handler, identified by the **Catch** command. This is known as throwing an exception. If no error occurs, execution continues with the next Caché Basic statement after the **End Try** statement.

An exception may occur as a result of a runtime error, such as attempting to divide by 0, or it may be explicitly propagated by issuing a **Throw** command.

A **Try** block must be immediately followed by a **Catch** block. The paired **Try** and **Catch** are terminated by an **End Try** statement.

Examples

In the following examples, the **Try** code block is executed. It attempts to set the local variable *a*. In the first example, the code completes successfully, and the **Catch** is skipped over. In the second example, the code fails an Err error indicating division by zero, and execution is passed to the **Catch** command.

Try succeeds:

```
Try
  SET x=2
  PRINTLN "about to divide by ",x
  SET a=7/x
  PRINTLN "Success: the result is ",a
Catch myvar
  PRINTLN "this is the exception handler"
  PRINTLN "Error number: ",Err.Number
  PRINTLN "Error is: ",Err.Description
  PRINTLN "Error code: ",myvar.Code
End Try
PRINTLN "this is where the code falls through"
```

Try fails:

```
Try
  SET x=0
  PRINTLN "about to divide by ",x
  SET a=7/x
  PRINTLN "Success: the result is ",a
Catch myvar
  PRINTLN "this is the exception handler"
  PRINTLN "Error number: ",Err.Number
  PRINTLN "Error is: ",Err.Description
  PRINTLN "Error code: ",myvar.Code
End Try
PRINTLN "this is where the code falls through"
```

See Also

- Catch command
- Throw command

- Err object

TStart

Marks the beginning of a transaction.

```
TStart
```

Arguments

The **TStart** statement does not have any arguments.

Description

TStart marks the beginning of a transaction. Following **TStart**, database operations are journaled to enable a subsequent **TCommit** or **TRollback** statement.

Any locks issued within a transaction will be held until the end of the transaction even if the lock is released.

Examples

The following example illustrates the use of the **TStart** statement:

```
TStart
If StorePerson(personobject) Then
 TCommit
Else
 TRollback
End If
```

See Also

- TCommit Statement
- TRollback Statement

While...Wend

Executes a series of statements as long as a given condition is true.

```
While condition
    [statements]
Wend
```

Arguments

The **While...Wend** statement syntax has these parts:

conditions	Expression that evaluates to **True** or **False**.
statements	One or more statements executed while condition is True.

Description

If condition is **True**, all statements in 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**.

Examples

The following example illustrates use of the **While...Wend** statement:

```
Dim Counter
Counter = 0                ' Initialize variable.
While Counter < 20         ' Test value of Counter.
   Counter = Counter + 1   ' Increment Counter.
   Print Counter
Wend                       ' End While loop when Counter > 19.
```

Notes

The **Do...Loop** statement provides a more structured and flexible way to perform looping.

See Also

- Do...Loop Statement
- Exit Statement
- For Each...Next Statement
- For...Next Statement

With

Executes a series of statements on a single object.

```
With object
   statements
End With
```

Arguments

The **With** statement requires the following arguments:

object	An expression that resolves to an object reference. *object* may be a function call that returns an object reference, or a subscripted variable that contains an object reference.
statements	One or more statements to be executed on the object.

Description

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** block code, referring to the object once instead of referring to it with each property assignment.

The *object* object reference is evaluated upon entering the **With** block, and is not reevaluated within the **With** block. Therefore, you cannot use a single **With** statement to affect a number of different objects. Changing the *object* variable value within the **With** block is permitted, but does not change which object is used for anonymous references within the **With** block.

While property manipulation is an important aspect of **With** functionality, it is not the only use. Any legal code can be used within a **With** block.

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.

A Goto statement cannot be used to enter the body of a **With** block, or into a nested inner **With** block. You can, however, issue a **Goto** within a **With** block to a label within that block, or to an outer **With** block label, or to a label outside the **With** block.

Examples

The following example illustrates use of the **With** statement to assign values to several properties of the same object.

```
With myPerson
   .City = "Cambridge"
   .State = "MA"
   .Street = "One Memorial Drive"
End With
```

Caché Basic Functions

Abs

Returns the absolute value of a number.

```
Abs(number)
```

Arguments

The *number* argument can be any valid numeric expression. If *number* is an uninitialized variable or a non-numeric value, **Abs** returns 0 (zero).

Description

The absolute value of a number is its unsigned magnitude. For example, **Abs**(-1) and **Abs**(1) both return 1. **Abs** removes signs, and leading and trailing zeros from *number*.

Examples

The following example uses the **Abs** function to compute the absolute value of a number:

```
Println Abs(0050.300)  'Returns 50.3
Println Abs(-50.3)     'Returns 50.3
Println Abs(+50.3)     'Returns 50.3
Println Abs(0)         'Returns 0
Println Abs(-0)        'Returns 0
```

See Also

- Sgn function

Asc

Returns the ANSI character code corresponding to the first character in a string.

```
Asc(string)
```

Arguments

The *string* argument is any valid string expression. A number is treated as a string expression. If the *string* contains no characters (an empty string), -1 is returned.

Description

The **Asc** function takes a character and returns the corresponding ANSI code. The **Chr** function takes an ANSI code and returns the corresponding character.

In ObjectScript, the **$ASCII** function performs the same operation.

Examples

In the following example, **Asc** returns the ANSI character code of the first character of each string:

```
Println Asc("A")         ' Returns 65
Println  Asc("a")        ' Returns 97
Println  Asc("Apple")    ' Returns 65
Println Asc(">")         ' Returns 62
Println Asc(Chr(959+1))  ' Returns 960
Println Asc(12345)       ' Returns 49
Println Asc("")          ' Returns -1
```

See Also

- Basic: Chr function
- ObjectScript: $ASCII function

Atn

Returns the arctangent of a number.

```
Atn(number)
```

Arguments

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

Description

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.

Examples

The following example returns the arctangents of the integers from -4 through 4:

```
For x = -4 TO 4
Println "Arctangent of ",x," is: ",Atn(x)
Next
```

The following example uses **Atn** to calculate the value of pi:

```
Dim pi
pi = 4 * Atn(1)   ' Calculate the value of pi.
Println "pi is: ",pi
```

Notes

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

See Also

- Cos function
- Sin function
- Tan function
- Derived Math Functions

Case

Compares a target to cases and returns the value associated with the first matching case.

```
Case(target,case:rvalue,case:rvalue,...,:default)
```

Arguments

target	A value, variable, or expression to be compared with the *case* arguments.
case	A value, variable, or expression, the value of which is matched with the value of *target*.
rvalue	The value to be returned upon a successful match of *target* and *case*.
default	*Optional* — The value to be returned if no *case* matches *target*.

Description

The **Case** function compares *target* to a list of cases (literals or expressions), and returns the *rvalue* associated with the first matching *case* value. An unlimited number of *case*:*rvalue* pairs can be specified. Cases are matched in the order specified (left-to-right); matching stops when the first exact match is encountered.

If there is no matching case, the default is returned. If there is no matching case and no default is specified, an error is returned.

Arguments

target

CASE evaluates the *target* expression once, then matches the result to each *case* value in left-to-right order.

case

A *case* can be a literal or an expression; matching of literals is substantially more efficient than matching expressions, because literals can be evaluated at compile time. Each *case* must be paired with an *rvalue*. An unlimited number of *case* and *rvalue* pairs may be specified.

rvalue

An *rvalue* can be a literal or an expression. Every *rvalue* is associated with a specific *case* as a pair joined with a colon (:) and separated from other pairs by a comma (,). *rvalue* is the value returned when there is an exact match of the *target* value with its associated *case* value. Only the first exact match encountered (in left-to-right order) returns an *rvalue*.

default

A *default* argument can be a literal or an expression. The *default* is specified like a *case*:*rvalue* pair, except that there is no *case* specified between the comma separator and the colon. The *default* is always the final argument specified in a **CASE** function. The *default* value is the value returned if no exact match occurs.

Examples

The following **Case** example takes a numeric input and writes out the appropriate explanatory string:

```
input "Input a number 1-3: ",x
multi=CASE(x,1:"single",2:"double",3:"triple",:"input error")
PrintLn multi
```

See Also

- If...Then...Else statement

Chr

Returns the character corresponding to the specified ANSI character code.

```
Chr(charcode)
```

Arguments

The *charcode* argument is a decimal integer that identifies a character. For 8-bit characters, the value in *charcode* must evaluate to a positive integer in the range 0 to 255. For 16-bit characters, specify integers in the range 256 through 65534.

Description

The **Chr** function takes an ANSI code and returns the corresponding character. The **Asc** function takes a character and returns the corresponding ANSI code.

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **Chr**(10) returns a linefeed character.

The Caché Basic **Chr** function returns a single character. The corresponding ObjectScript **$CHAR** function can return multiple characters by specifying a comma-separated list of ASCII codes.

Examples

The following example uses the **Chr** function to return the character associated with the specified character code:

```
Println Chr(65)    ' Returns A.
Println Chr(97)    ' Returns a.
Println Chr(37)    ' Returns %.
Println Chr(62)    ' Returns >.
Println Chr(960)   ' Returns the symbol for pi.
```

See Also

- Basic: Asc function
- ObjectScript: $CHAR function

Cos

Returns the cosine of an angle.

```
Cos(number)
```

Arguments

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Description

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.

Examples

The following example uses the **Cos** function to return the cosine of an angle:

```
Dim MyAngle
MyAngle = 1.3           ' Define angle in radians.
Println Cos(MyAngle)    ' Calculate cosine.
```

The following example uses the **Cos** function to return the secant of an angle:

```
Dim MyAngle, MySecant
MyAngle = 1.3                ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)  ' Calculate secant.
Println MySecant
```

See Also

- Atn function
- Sin function
- Tan function
- Derived Math Functions

Date

Returns the current system date.

```
Date
```

Arguments

none

Description

The **Date** function returns the current date in a format such as the following:

```
mm/dd/yyyy
```

The exact display format depends on your system configuration. Leading zeros are displayed. The year is displayed as four digits.

Examples

The following example uses the **Date** function to return the current system date:

```
Dim MyDate
MyDate = Date
Println MyDate
```

See Also

- Basic: Now function, Time function
- ObjectScript: $HOROLOG special variable
- SQL: NOW function

DateAdd

Returns a date to which a specified time interval has been added.

```
DateAdd(interval,number,date)
```

Arguments

interval	A string expression code that specifies the interval type you want to add, specified as a quoted string. The table of available codes is shown below.
number	A numeric expression that is the number of intervals you want to add. The numeric expression can either be positive to add intervals, or negative to subtract intervals.
date	Variable name or literal representing the date to which the specified interval is added (or subtracted). The date can optionally have a time component; if not specified, the time defaults to 00:00:00.

The **DateAdd** function returns the calculated date in the following format:

```
mm/dd/yyyy
```

Leading zeros are displayed. The year is displayed as four digits.

Description

The *interval* argument can have the following values:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of Year (this adds the specified number of days to *date*; same as “d”).
d	Day
w	Weekday (this adds the specified number of days to *date*; same as “d”).
ww	Week (this adds the specified number of weeks to *date*).
h	Hour
n	Minute (note this is “n”, not “m”)
s	Second

You can use the **DateAdd** function to add or subtract a specified interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from a given date, or a date 100 hours earlier than a given date. To add days to *date*, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function computes the varying number of days in different months (including leap years), and avoids returning an invalid date.

Examples

The following example adds one of each date interval unit to January 31, 2005:

```
NewDay = DateAdd("d",1,"31-Jan-2005")
NewWDay = DateAdd("w",1,"31-Jan-2005")
NewWeek = DateAdd("ww",1,"31-Jan-2005")
NewMonth = DateAdd("m",1,"31-Jan-2005")
NewQuarter = DateAdd("q",1,"31-Jan-2005")
NewYDay = DateAdd("y",1,"31-Jan-2005")
NewYear = DateAdd("yyyy",1,"31-Jan-2005")
Println NewDay
Println NewWDay
Println NewWeek
Println NewMonth
Println NewQuarter
Println NewYDay
Println NewYear
```

In the case of adding one month to 31-Jan-2005, **DateAdd** returns 02/28/2005, not 02/31/2005. If date is 31-Jan-2004, it returns 02/29/2005, because 2004 is a leap year. If the calculated date would precede the year 100, an error occurs.

The following example adds a time interval large enough to increment the specified date:

```
NewHour = DateAdd("h",27, "31-Jan-2005")
NewMin = DateAdd("n",1545, "31-Jan-2005")
NewSec = DateAdd("s",91522, "31-Jan-2005")
Println NewHour
Println NewMin
Println NewSec
```

Note that the values returned contain both a date and a time component.

See Also

- DateDiff function
- DatePart function

DateConvert

Converts dates between internal and external formats.

```
DateConvert(date,vbToInternal)
DateConvert(date,vbToExternal)
```

Arguments

date	The date to be converted. An external date is represented as a string, such as “10–22–1980”. An internal date is represented as a five-digit integer, which is the first part of the Caché $HOROLOG ($H) date/time format.
vbToInternal	This keyword specifies converting an external date to internal ($H) format.
vbToExternal	This keyword specifies converting an internal date ($H format) to external date format.

Description

If you specify a date/time value, **DateConvert** ignores the time portion.

The **DateConvert** function returns an external date in the following format:

```
mm/dd/yyyy
```

Leading zeros are displayed. The year is displayed as four digits.

The **DateConvert** function returns an internal date/time in the following format:

```
ddddd
```

Where “d” is the date count (number of days since 12/31/1840). For further details, see $HOROLOG in the *Caché ObjectScript Reference*.

An omitted year value defaults to 2000; the two-digit year defaults are 2000 through 2029 (for 00 through 29) and 1930 through 1999 (for 30 through 99).

Examples

The following example takes an external date/time value, converts the date part to an internal format ($HOROLOG) value, then converts this internal value back to an external format date.

```
Dim GetDate,InDate, ExDate
GetDate = "1-12-1953 11:45:23"
Println GetDate
InDate = DateConvert(GetDate,vbToInternal)
Println InDate
ExDate = DateConvert(InDate,vbToExternal)
Println ExDate
```

The values printed are as follows:

```
1-12-1953 11:45:23
40919
01/12/1953
```

See Also

- Basic: DateTimeConvert function
- Basic: TimeConvert function

- ObjectScript: $HOROLOG special variable

DateDiff

Returns the number of intervals between two dates.

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

Arguments

interval	A string expression code that specifies the interval you want to use to calculate the difference between *date1* and *date2*. Specified as a quoted string. A list of these *interval* codes is provided below.
date1, *date2*	Two date expressions. The two dates you want to use in the calculation. These date expressions can optionally also include a time component. If the time is omitted, it defaults to 00:00:00.
firstdayofweek	*Optional* — Constant that specifies the day of the week. If not specified, Sunday is assumed. A list of the available constants is provided below.
firstweekofyear	*Optional* — 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. A list of the available constants is provided below.

Description

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. **DateDiff** returns a positive integer for the number of intervals if *date1* is earlier than *date2*; otherwise it returns a negative integer for the number of intervals. If both dates are the same, or if the time between them is less than the specified interval, **DateDiff** returns zero (0).

Intervals are calculated from the specified unit itself. Thus a year interval is determined by whether the two year dates differ, not by how many days have elapsed. Similarly, a day interval is determined by whether the two dates differ, not by how many hours have elapsed.

The *interval* argument can have the following values:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of Year
d	Day
w	Weekday (number of seven-day units)
ww	Week (number of calendar weeks)
h	Hour
n	Minute
s	Second

To calculate the number of days between *date1* and *date2*, you can use either Day ("d") or Day of Year ("y").

To calculate the number of weeks between *date1* and *date2* you can use Weekday ("w") or Week ("ww"). 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 does not count *date1*, even if it does fall on a Sunday.

The *firstdayofweek* argument affects calculations that use the "w" and "ww" interval symbols. The *firstdayofweek* argument can have the following values:

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

The *firstweekofyear* argument can have the following values:

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

The Year ("yyyy") interval calculates number of years based on the year date, not the number of elapsed days. Thus, 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.

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 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.

Examples

The following example uses the **DateDiff** function to display the number of days between a given date and today:

```
DiffADate =  DateDiff("d","11/12/1953",Date)
Print "Days to the present day: "
Println DiffADate
```

The following example calculates the number of each date interval unit between November 12, 1953 and November 1, 2005:

```
NewDay = DateDiff("d","11/12/1953","11/1/2005")
NewWeek = DateDiff("w","11/12/1953","11/1/2005")
NewMonth = DateDiff("m","11/12/1953","11/1/2005")
NewQuarter = DateDiff("q","11/12/1953","11/1/2005")
NewYDay = DateDiff("y","11/12/1953","11/1/2005")
NewYear = DateDiff("yyyy","11/12/1953","11/1/2005")
Println NewDay
Println NewWeek
Println NewMonth
Println NewQuarter
Println NewYDay
Println NewYear
```

The following example calculates the number of each days between January 1 and March 1 on a leap year (2004) and a non-leap year (2005):

```
LeapDays = DateDiff("d","1/1/2004","3/1/2004")
NLeapDays = DateDiff("d","1/1/2005","3/1/2005")
Println LeapDays
Println NLeapDays
```

As one would expect, the difference is 60 days in leap years, and 59 days in non-leap years.

The following example calculates the number of time intervals between two successive days. Note that if the time is not specified, it defaults to 00:00:00:

```
NumH = DateDiff("h","1/1/2004","1/2/2004")
NumHNoon = DateDiff("h","1/1/2004","1/2/2004 12:00:00")
NumMin = DateDiff("n","1/1/2004","1/2/2004")
NumMinNoon = DateDiff("n","1/1/2004","1/2/2004 12:00:00")
NumSec = DateDiff("s","1/1/2004","1/2/2004")
NumSecNoon = DateDiff("s","1/1/2004","1/2/2004 12:00:00")
Println NumH
Println NumHNoon
Println NumMin
Println NumMinNoon
Println NumSec
Println NumSecNoon
```

See Also

- DateAdd function
- DatePart function

DatePart

Returns the specified part of a given date.

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

Arguments

interval	A string expression code that is the interval of time you want to return. See below for code values.
date	Date expression you want to evaluate, specified as a quoted string.
firstdayofweek	*Optional* — Constant that specifies the day of the week. If not specified, Sunday is assumed. See below for values.
firstweekofyear	*Optional* — 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. See below for values.

Description

You can use the **DatePart** function to evaluate a date and return a specific interval as an integer value. For example, you might use **DatePart** to calculate the day of the week or the number of days since the start of the year.

The *interval* argument can have the following values:

Setting	Description
yyyy	Year date
q	Quarters since beginning of year
m	Month date; number of months since beginning of year
y	Day of Year; number of days since beginning of year.
d	Day date; number of days since beginning of month.
w	Weekday (day of the week, with Sunday counted as 1)
ww	Weeks since beginning of year
h	Hour (defaults to 1).
n	Minute (defaults to 0).
s	Second (defaults to 0).

The *firstdayofweek* argument can have the following values:

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

The *firstweekofyear* argument can have the following values:

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

You must specify a *firstdayofweek* argument value in order to specify a *firstweekofyear* argument value. The *firstdayofweek* argument affects calculations that use the "w" and "ww" intervals.

If date is a date literal, the specified year becomes a permanent part of that date. However, if date is enclosed in 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.

Examples

The following example takes a date and displays the corresponding interval counts:

```
NewDay = DatePart("d","30-Nov-2005")
NewWDay = DatePart("w","30-Nov-2005")
NewWeek = DatePart("ww","30-Nov-2005")
NewMonth = DatePart("m","30-Nov-2005")
NewQuarter = DatePart("q","30-Nov-2005")
NewYDay = DatePart("y","30-Nov-2005")
NewYear = DatePart("yyyy","30-Nov-2005")
NewHour = DatePart("h","30-Nov-2005")
NewMin = DatePart("n","30-Nov-2005")
NewSec = DatePart("s","30-Nov-2005")
Println NewDay
Println NewWDay
Println NewWeek
Println NewMonth
Println NewQuarter
Println NewYDay
Println NewYear
Println NewHour
Println NewMin
Println NewSec
```

The following example shows the effects of the *firstdayofweek* argument:

```
MyDay0 = DatePart("w","11/1/2005",vbUseSystem)
MyDay1 = DatePart("w","11/1/2005",vbSunday)
MyDay2 = DatePart("w","11/1/2005",vbMonday)
MyDay3 = DatePart("w","11/1/2005",vbTuesday)
MyDay4 = DatePart("w","11/1/2005",vbWednesday)
MyDay5 = DatePart("w","11/1/2005",vbThursday)
MyDay6 = DatePart("w","11/1/2005",vbFriday)
MyDay7 = DatePart("w","11/1/2005",vbSaturday)
Println "Day is: ",MyDay0," Week begins System Default"
Println "Day is: ",MyDay1," Week begins Sunday"
Println "Day is: ",MyDay2," Week begins Monday"
Println "Day is: ",MyDay3," Week begins Tuesday"
Println "Day is: ",MyDay4," Week begins Wednesday"
Println "Day is: ",MyDay5," Week begins Thursday"
Println "Day is: ",MyDay6," Week begins Friday"
Println "Day is: ",MyDay7," Week begins Saturday"
```

Nov. 1, 2005 is a Tuesday. `DatePart("w","11/1/2005",vbTuesday)` returns 1.

The following example returns the week of the year count for February 29, 2008, based on different *firstdayofweek* and *firstweekofyear* argument values:

```
Println "Week is: ",DatePart("ww","2/29/2008",vbUseSystem,vbUseSystem)
Println "Week is: ",DatePart("ww","2/29/2008",vbThursday,vbUseSystem)
Println "Week is: ",DatePart("ww","2/29/2008",vbUseSystem,vbFirstJan1)
Println "Week is: ",DatePart("ww","2/29/2008",vbThursday,vbFirstJan1)
Println "Week is: ",DatePart("ww","2/29/2008",vbUseSystem,vbFirstFourDays)
Println "Week is: ",DatePart("ww","2/29/2008",vbThursday,vbFirstFourDays)
Println "Week is: ",DatePart("ww","2/29/2008",vbUseSystem,vbFirstFullWeek)
Println "Week is: ",DatePart("ww","2/29/2008",vbThursday,vbFirstFullWeek)
```

Note that both *firstweekofyear* and *firstdayofweek* can affect the week of the year count.

See Also

- DateAdd function
- DateDiff function

DateSerial

Returns the date for a specified year, month, and day.

```
DateSerial(year,month,day)
```

Arguments

year	A four-digit integer between 1841 and 9999, inclusive, a two-digit integer, or a numeric expression that evaluates to an integer within these ranges.
month	A positive or negative integer or a numeric expression that evaluates to an integer. A *month* value of 0 or the empty string ("") is interpreted as the last month of the previous year. A negative *month* value backs up the specified number of months from the last month of the previous year. Thus -1 is the 11th month of the previous year.
day	A positive or negative integer or a numeric expression that evaluates to an integer. A *day* value of 0 or the empty string ("") is interpreted as the last day of the previous month. A negative *day* value backs up the specified number of days from the last day of the previous month. Thus, a day value of -1 is interpreted as the day before the last day of the previous month.

Description

DateSerial takes the input arguments and generates a valid date in the format:

```
mm/dd/yyyy
```

The range of numbers for each **DateSerial** argument can be an exact date value or a relative date value. A relative date value is an integer value outside the accepted range for the unit; that is, 1–31 for days and 1–12 for months. In this case, **DateSerial** uses these numbers to calculate a valid date. Any numeric expression can be used to represent some number of days, months, or years before or after a certain date.

Year values between 0 and 99, inclusive, are interpreted as the years 1900–1999. The empty string ("") is interpreted as the year 1900. For all other year arguments, use a complete four-digit year (for example, 2005). The earliest allowed *year* value is 1841.

Examples

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) of two months before August (8 – 2) of 10 years before 1990 (1990 – 10); in other words, May 31, 1980.

```
Dim MyDate1, MyDate2
MyDate1 = DateSerial(1970, 1, 1)
MyDate2 = DateSerial(1990 - 10, 8 - 2, 1 - 1)
Println MyDate1  ' Returns 01/01/1970 (January 1, 1970)
Println MyDate2  ' Returns 05/31/1980
```

The following example uses *month* values of 0, the empty string (""), and negative numbers:

```
Println DateSerial(2009,"",3)  ' Returns 12/03/2008
Println DateSerial(2009,0,3)   ' Returns 12/03/2008
Println DateSerial(2009,-1,3)  ' Returns 11/03/2008
Println DateSerial(2009,-2,3)  ' Returns 10/03/2008
```

The following example uses *day* value that is not valid for the specified month. **DateSerial** is aware of leap year values and adjusts the month accordingly:

```
Println DateSerial(2009,2,29)  ' Returns 03/01/2008
```

The following example uses *day* and *month* values that are larger than the number of days in the specified month and monthsd in a year. **DateSerial** adjusts the day, month, and year accordingly:

```
Println DateSerial(2009,13,40)  ' Returns 02/09/2010
```

Notes

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. However, if any single argument is outside the range -32,768 to 32,767, or if the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates (12/31/1840 through 12/31/9999), an error occurs.

See Also

- Date function
- Day function
- Month function
- Now function
- TimeSerial function
- Weekday function
- Year function

DateTimeConvert

Converts date/time between internal and external formats.

```
DateTimeConvert(datetime,vbToInternal)
DateTimeConvert(datetime,vbToExternal)
```

Arguments

datetime	The date and time to be converted. An external date/time is represented as a string, such as “10–22–1980 12:35:56”. An internal date/time is represented by the Caché $HOROLOG ($H) date/time format: two five-digit integer values separated by a comma.
vbToInternal	This keyword specifies converting an external date/time to internal ($H) format.
vbToExternal	This keyword specifies converting an internal date/time ($H format) to external date and time format.

Description

The **DateTimeConvert** function returns an external date/time in the following format:

```
mm/dd/yyyy hh:mm:ss
```

Leading zeros are displayed. The year is displayed as four digits.

The **DateTimeConvert** function returns an internal date/time in the following format:

```
ddddd,sssss.ff
```

Where “ddddd” is the date count (number of days since 12/31/1840), “sssss” is the time count (number of elapsed seconds in the specified day), and “ff” is optional fractional seconds. Fractional seconds are preserved in converting from external to internal format; fractional seconds are truncated when converting from internal to external format. For further details, see $HOROLOG in the *Caché ObjectScript Reference*.

An omitted year value defaults to 2000; the two-digit year defaults are 2000 through 2029 (for 00 through 29) and 1930 through 1999 (for 30 through 99).

An omitted time value defaults to 00:00:00.

Examples

In the following example the **DateTimeConvert** function returns a date/time in internal format.

```
Dim InDateTime
InDateTime = DateTimeConvert("Nov 11 1953 12:35:00",vbToInternal)
Println InDateTime
```

The following example takes an external date/time value with fractional seconds, converts it to an internal format ($HOROLOG) value, then converts this internal value back to an external format date and time.

```
Dim GetDate,InDate, ExDate
GetDate = "1-12-1953 11:45:23.99"
Println GetDate
InDate = DateTimeConvert(GetDate,vbToInternal)
Println InDate
ExDate = DateTimeConvert(InDate,vbToExternal)
Println ExDate
```

The values printed are as follows:

```
1-12-1953 11:45:23.99
40919,42323.99
01/12/1953 11:45:23
```

See Also

- Basic: DateConvert function
- Basic: TimeConvert function
- ObjectScript: $HOROLOG special variable

Day

Returns the day of the month as an integer between 1 and 31, inclusive.

```
Day(date)
```

Arguments

The *date* argument is any expression that represents a date as a string.

Description

The **Day** function locates and returns the numeric day portion of a date string as an integer. It performs no range validation on this number. The **Day** function accepts blanks, slashes (/), hyphens (-), or commas (,) (in any combination) as date component separators. Leading zeros and plus or minus signs may be included or omitted in the input string; leading zeros and signs are omitted from the output integer. The **Day** function locates the day portion in either of two ways:

- In American format, the month precedes the day. For example, “9/27/2005” or “September 27, '05.” In this format, the **Day** function identifies the day portion by position. It does not parse the month or year components of the date string. These can be any alphanumeric value, and can include or omit punctuation characters such as periods or apostrophes. The year component may be 4-digits, less than 4 digits, or omitted.
- In European written format, the day precedes the name of the month. For example, “27 September 2005” or “27 Sept” In this case, the month name is validated; the first three letters must correspond to a valid month name. Validation is not case-sensitive.

If the **Day** function is unable to identify a day portion of a string, it returns 0.

Examples

The following example uses the **Day** function to return the current day of the month:

```
Dim MyDay
MyDay = Day(Date)
Print MyDay
```

The following examples use the **Day** function to obtain the day of the month from a specified date:

```
Dim MyDay
MyDay = Day("09-19-2005")  'MyDay contains 19.
Print MyDay

Dim MyDay
MyDay = Day("09/19/05")  'MyDay contains 19.
Print MyDay

Dim MyDay
MyDay = Day("Sept 19, 2005")  'MyDay contains 19.
Print MyDay

Dim MyDay
MyDay = Day("19 October")  'MyDay contains 19.
Print MyDay

Dim MyDay
MyDay = Day("19 Feb")  'MyDay contains 19.
Print MyDay
```

See Also

- Basic: Date function, Hour function, Minute function, Month function, Now function, Second function, Weekday function, Year function

- ObjectScript: $ZDATE function
- SQL: DAYOFMONTH function

Derived Math Functions

The following non-intrinsic math functions can be derived from the intrinsic math functions:

Description

Caché Basic supplies four trigonometric functions: **Sin** (sine), **Cos** (cosine), **Tan** (tangent), and **Atn** (arctangent); two logarithmic functions: **Log** (natural e logarithm) and **Exp** (e exponential); the **Sqr** (square root) function and the **Sgn** (sign) function. From these many other functions and constants can be derived.

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 Tangent	HArctan(X) = Log((1 + X) / (1 - X)) / 2
Inverse Hyperbolic Secant	HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) +1) / X)
Inverse Hyperbolic Cotangent	HArccotan(X) = Log((X + 1) / (X - 1)) / 2
Base-10 Logarithm	Log10(X) = Log(X) / Log(10)
Logarithm to base N	LogN(X) = Log(X) / Log(N)

ObjectScript Equivalents

ObjectScript supplies the following nine trigonometric functions: $ZSIN sine function; $ZCOS cosine function; $ZARCSIN inverse (arc) sine function; $ZARCCOS inverse (arc) cosine function; $ZTAN tangent function; $ZARCTAN inverse (arc) tangent function; $ZCOT cotangent function; $ZSEC secant function; and $ZCSC cosecant function.

ObjectScript supplies the following three logarithmic functions: $ZEXP e exponential function; $ZLN natural logarithm function; and $ZLOG base-10 logarithm function.

ObjectScript supplies the following two exponential functions: $ZPOWER exponent function; and $ZSQR square root function.

See Also

- Atn function
- Cos function
- Exp function
- Log function
- Sgn function
- Sin function
- Sqr function
- Tan function

Exists

Returns the existence status of variables and their array subnodes.

```
Exists(varname)
```

Arguments

varname	Name of a variable to test for existence, and/or the presence of array subnodes.

Description

The **Exists** function returns an integer code indicating whether a variable is defined (1) or not (0). If the variable is an array, **Exists** returns an integer code indicating that the specified node's value is undefined but the node has defined subnodes (2), or that the specified node's value is defined and the node has defined subnodes (3).

These values can also be represented by the following constants: 0 = vbUndef; 1 = vbHasValue; 2 = vbHasArray. The 3 value is equivalent to vbHasValue and vbHasArray. Refer to the Node Constants page of this manual.

The *varname* argument must contain a variable, not an expression. For example, ME is an expression, so `Exists(ME)` generates a compile error. However, `Exists(ME.Property)` is a valid use of **Exists**.

Examples

The following example demonstrates the use of the **Exists** function:

```
Println "x is: ",Exists(x)  ' x is undefined
x = 7
Println "x is: ",Exists(x)  ' x is defined
x(1) = 6
Println "x(1) is: ",Exists(x)  ' x & x(1) defined
y(1) = 55
Println "y(1) is: ",Exists(y)  ' y(1) defined, y not
```

The above example returns (in sequence): 0, 1, 3, 2.

The following example further demonstrate use of the **Exists** function with array nodes:

```
' Erase previously existing data
Erase ^User.TestData

' Create some demonstration global data
^User.TestData(1)="data"    ' Node 1 is defined but no subnodes
^User.TestData(2,1)="data"  ' Node 2 is not defined but has subnodes
^User.TestData(3)="data"    ' Node 3 is defined and has subnodes
^User.TestData(3,1)="data"

Status = Exists(^User.TestData(1,1)) ' prints vbUndef  0
Println Status," Undefined subnode"
Status = Exists(^User.TestData(1)) ' prints vbHasValue  1
Println Status," Defined node without subnodes"
Status = Exists(^User.TestData(2,1)) ' prints vbHasValue  1
Println Status," Defined subnode without subnodes"
Status = Exists(^User.TestData(2)) ' prints vbHasArray  2
Println Status," Valueless node with defined subnode(s)"
Status = Exists(^User.TestData(3)) ' prints  3,
                                   ' (vbHasValue + vbHasArray)
Println Status," Defined node with defined subnode(s)"
```

See Also

- Node Constants

Exp

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

```
Exp(number)
```

Arguments

The *number* argument can be any valid numeric expression. On a Windows system, if the value of *number* is greater than 335, a runtime error occurs; if the value of *number* is less than -295, **Exp** returns zero (0).

Description

The **Exp** function takes the natural log constant e and raises it to the power specified by the *number* argument. The constant e (**Exp(1)**) is approximately 2.718282.

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

In ObjectScript, the corresponding function is $ZEXP.

Examples

The following example uses the **Exp** function to calculate e raised to the power of each of the integers -10 through 10:

```
For x = -10 To 10
Println "Natural log of ",x," = ",Exp(x)
Next
```

The following example uses the **Exp** function to return the hyperbolic sine of an angle:

```
Dim MyAngle, MyHSin  ' Define angle in radians.
MyAngle = 1.3        ' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
Println MyHSin
```

See Also

- Log function
- Derived Math Functions

Fix

Returns the integer portion of a number.

```
Fix(number)
```

Arguments

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

Description

Fix removes the fractional part of *number* and returns the resulting integer value. The **Fix** and Int functions are almost functionally identical:

For positive values, both **Fix** and **Int** truncate *number*. If you wish to round a number to the nearest integer, use the **Round** function.

For negative values, **Fix** returns the first negative integer greater than or equal to *number*. **Int** returns the first negative integer less than or equal to *number*. For example, **Fix** converts -8.4 to -8 and **Int** converts -8.4 to -9.

Both **Fix** and **Int** remove leading zeros and plus signs from *number*.

Examples

The following examples illustrate how the **Fix** and **Int** functions return integer portions of numbers:

```
Println Int(99.8)      ' Returns 99.
Println Fix(99.8)      ' Returns 99.
Println Int(+99.20)    ' Returns 99.
Println Fix(+0099.2)   ' Returns 99.
Println Int(0.00)      ' Returns 0.
Println Fix(0.00)      ' Returns 0.
Println Int(-99.8)     ' Returns -100.
Println Fix(-99.8)     ' Returns -99.
Println Int(-99.2)     ' Returns -100.
Println Fix(-99.2)     ' Returns -99.
```

See Also

- Abs function
- Int function
- Round function

Hex

Returns a string representing the hexadecimal value of a number.

```
Hex(number)
```

Arguments

The *number* argument is any valid expression that resolves to a positive or negative number. If *number* is a decimal fraction, it is truncated to a whole number before being evaluated.

Description

The **Hex** function converts a number from decimal (base 10) to hexadecimal (base 16). To convert a number from hexadecimal to decimal, represent hexadecimal numbers directly by preceding numbers in the proper range with &H For example, &H10 is the hexadecimal notation for decimal 16.

If Number Is	Hex Returns
Empty	Zero (0).
Any other number	Up to eight hexadecimal characters.

Examples

The following example uses the **Hex** function to return the hexadecimal value of a decimal (base-10) number:

```
Println Hex(0)     ' Returns 0.
Println Hex(4)     ' Returns 4.
Println Hex(10)    ' Returns A.
Println Hex(16)    ' Returns 10.
Println Hex(459)   ' Returns 1CB.
```

The following example uses the &H prefix to return the decimal (base-10) value for a hexadecimal number:

```
Println &H000  ' Returns 0.
Println &H4    ' Returns 4.
Println &HA    ' Returns 10.
Println &H10   ' Returns 16.
```

See Also

- Oct function

Hour

Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

```
Hour(time)
```

Arguments

The time argument is any expression that can represent a time. This includes a time value such as “12:30” or “1:27:55”, a time/date value such as “11/12/1999 12:33:00”, or a date value such as “11/12/1999”. If only a date is specified, the time defaults to 00:00:00. Fractional seconds are permitted, but ignored; they are truncated, not rounded.

Examples

The following example uses the **Hour** function to obtain the hour from the current time:

```
Dim MyTime, MyHour
MyTime = Now
MyHour = Hour(MyTime) ' MyHour contains the number representing
                      ' the current hour.
Println MyHour
```

The following example returns an hour value of 13:

```
Dim MyHour
MyHour = Hour("13:59:59.999")
Println MyHour
```

See Also

- Day function
- Minute function
- Now function
- Second function
- Time function

Increment

Atomically increments the value of a variable and returns the new value.

```
Increment(varname[,change])
```

Arguments

varname	The name of the variable to be incremented (or decremented).
change	*Optional* — A numeric that specifies by how much the value should be incremented. The value of *change* can be negative for decrements. This value can be a fractional number. If specified as 0 (zero), no increment or decrement occurs. If not specified, *varname* is incremented by 1.

Description

The virtual machine ensures that during the increment the variable is locked and after the increment unlocked. Because of the atomic nature of this function, this operation is very efficient especially in network environments.

Examples

The following example demonstrates the use of the **Increment** function:

```
^PersonRecords = 1000
NewPersId = Increment(^PersonRecords)
Println NewPersId 'prints 1001
NewPersId = Increment(^PersonRecords, 10)
Println NewPersId 'prints 1011
```

The following example demonstrates the use of the **Increment** function to decrement a number:

```
countdown = 10
While countdown > 0
  Println countdown
  countdown = Increment(countdown,-1)
Wend
  Println "Blast off!"
```

The following example demonstrates the use of the **Increment** function with a fractional value to represent successive sevenths of a circle:

```
angle = 0
sevenths = 51.428572
While angle < 360
  angle = Increment(angle,sevenths)
  Println angle," degrees"
Wend
```

InStr

Returns the position of the first occurrence of one string within another.

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

Arguments

start	*Optional* — Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. The *start* argument is required if *compare* is specified.
string1	String expression being searched.
string2	String expression being searched for.
compare	*Optional* — Numeric value indicating the kind of comparison to use when evaluating substrings. See Description section for values. If omitted, a binary comparison is performed.

Note: The order of the arguments in the syntax for the **InStr** function is not the same as the **InStrRev** function syntax.

Description

The **InStr** function searches a string from left-to-right, and returns the location of the first occurrence of *string2* encountered. The location returned is a positive integer counting from left-to-right, with the first (leftmost) character of the string being location 1.

If the *start* value is equal to or less than the location of the first (leftmost) character of *string2*, **InStr** finds the first instance of *string2*, searching forewards from that point, and returns the location of the first (leftmost) character of *string2*. If the *start* value is greater than the location of the first (leftmost) character of *string2*, **InStr** returns 0. If the *start* value is omitted, **InStr** searches the entire string and returns the first *string2* location found. If the *start* value is greater than the *string1* length, **InStr** returns 0. If the *start* value is a negative number, **InStr** returns 0.

The *compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

The **InStr** function returns the following values:

If	InStr Returns
string1 is zero-length	0
string2 is zero-length	start
string2 is not found	0
string2 is found within string1	Position at which match is found
start > Len(string2)	0

Examples

The following examples use **InStr** to search a string:

```
Dim SearchString, SearchChar
SearchString ="XXpXXpXXPXXP"        ' String to search in.
SearchChar = "P"                    ' Search for "P".

' A textual comparison starting at position 4. Returns 6.
PRintln Instr(4, SearchString, SearchChar, 1)

' A binary comparison starting at position 1. Returns 9.
Println Instr(1, SearchString, SearchChar, 0)

' Comparison is binary by default (last argument is omitted).
Println Instr(SearchString, SearchChar)    ' Returns 9.

' A binary comparison starting at position 1. Returns 0 ("W" is not found).
Println Instr(1, SearchString, "W")
```

The following example shows the use of the *start* argument.

```
Println "-1: ",InStr(-1,"abcdefg","bc")  ' Returns 2
Println "0: ",InStr(0,"abcdefg","bc")    ' Returns 2
Println "1: ",InStr(1,"abcdefg","bc")    ' Returns 2
Println "2: ",InStr(2,"abcdefg","bc")    ' Returns 2
Println "3: ",InStr(3,"abcdefg","bc")    ' Returns 0
Println "6: ",InStr(6,"abcdefg","bc")    ' Returns 0
Println "7: ",InStr(7,"abcdefg","bc")    ' Returns 0
Println "8: ",InStr(8,"abcdefg","bc")    ' Returns 0
```

See Also

- InStrRev function

InStrRev

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

```
InStrRev(string1,string2[,start[,compare]])
```

Arguments

string1	String expression being searched.
string2	String expression being searched for.
start	*Optional* —An integer that sets the starting position for the reverse direction search. The *start* position is counted from left to right (counting from 1); the search is done from right to left. Thus you should specify a positive integer specifying a *start* position to the right of the expected location of *string2*. The *start* position can be the actual position of *string2*. To search the entire string, from right to left, specify -1. If *start* is omitted, -1 is the default.
compare	*Optional* — Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Description section for values.

Note: The order of the arguments in the syntax for the **InStrRev** function is not the same as the **InStr** function syntax.

Description

The **InStrRev** function searches a string from right-to-left, and returns the location of the first occurrence of *string2* encountered. The location returned is a positive integer counting from left-to-right, with the first (leftmost) character of the string being location 1. The location returned specifies the beginning (leftmost) character of *string2*.

If the *start* value is equal to or greater than the location of the last (rightmost) character of *string2*, **InStrRev** finds the first instance of *string2*, searching backwards from that point, and returns the location of the first (leftmost) character of *string2*. If the *start* value is less than the location of the last (rightmost) character of *string2*, **InStrRev** returns 0. If the *start* value is -1 or omitted, **InStrRev** searches the entire string and returns the first *string2* location found, searching backwards from the end of the string. If the *start* value is greater than the *string1* length, **InStrRev** returns 0. If the *start* value is a negative number other than -1, **InStrRev** returns 0.

The *compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

The **InStrRev** function returns the following values:

If	InStr Returns
string1 is zero-length	0
string2 is zero-length	start
string2 is not found	0
string2 is found within string1	Position at which match is found
start > Len(string2)	0

Examples

The following examples use **InStrRev** to search a string:

```
Dim SearchString, SearchChar, MyPos
SearchString ="XXpXXpXXPXXP"           ' String to search in.
SearchChar = "P"                       ' Search for "P".

' A binary comparison starting at position 10. Returns 9.
Println InstrRev(SearchString, SearchChar, 10, 0)

' A textual comparison starting at the last position. Returns 12.
Println InstrRev(SearchString, SearchChar, -1, 1)

' Comparison is binary by default (last argument is omitted). Returns 0.
Println InstrRev(SearchString, SearchChar, 8)
```

The following example shows the use of the *start* argument.

```
Println "-1: ",InStrRev("abcdefg","bc",-1)  ' Returns 2
Println "0: ",InStrRev("abcdefg","bc",0)    ' Returns 0
Println "1: ",InStrRev("abcdefg","bc",1)    ' Returns 0
Println "2: ",InStrRev("abcdefg","bc",2)    ' Returns 0
Println "3: ",InStrRev("abcdefg","bc",3)    ' Returns 2
Println "6: ",InStrRev("abcdefg","bc",6)    ' Returns 2
Println "7: ",InStrRev("abcdefg","bc",7)    ' Returns 2
Println "8: ",InStrRev("abcdefg","bc",8)    ' Returns 0
```

See Also

- InStr function

Int

Returns the integer portion of a number.

```
Int(number)
```

Arguments

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

Description

Int removes the fractional part of *number* and returns the resulting integer value. The **Int** and Fix functions are almost functionally identical:

For positive values, both **Int** and **Fix** truncate *number*. If you wish to round a number to the nearest integer, use the **Round** function.

For negative values, **Int** returns the first negative integer less than or equal to *number*. **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.

Both **Int** and **Fix** remove leading zeros and plus signs from *number*.

Examples

The following examples illustrate how the **Int** and **Fix** functions return integer portions of numbers:

```
Println Int(99.8)      ' Returns 99.
Println Fix(99.8)      ' Returns 99.
Println Int(+99.20)    ' Returns 99.
Println Fix(+0099.2)   ' Returns 99.
Println Int(0.00)      ' Returns 0.
Println Fix(0.00)      ' Returns 0.
Println Int(-99.8)     ' Returns -100.
Println Fix(-99.8)     ' Returns -99.
Println Int(-99.2)     ' Returns -100.
Println Fix(-99.2)     ' Returns -99.
```

See Also

- Abs function
- Fix function
- Round function

IsObject

Returns a value indicating whether an expression references a valid Automation object.

```
IsObject(expression)
```

Arguments

The *expression* argument can be any kind of expression (such as a numeric or string expression).

Description

Expression Evaluates To	IsObject Returns
valid Object Reference	1
invalid Object Reference	–1
not an Object Reference	0

IsObject returns a –1 value if *expression* is a reference to an invalid object. Invalid objects should not occur in normal operations; an invalid object could be caused, for example, by recompiling the class while instances of the class are active.

Examples

The following example uses the **IsObject** function to determine if an identifier represents an object variable:

```
o = New Sample.Person
Println IsObject(o)        ' Returns 1.
Println IsObject("hello")  ' Returns 0.
o = ""
Println IsObject(o)        ' Returns 0.
```

See Also

- New statement
- OpenId statement

Join

Returns a string created by joining a number of array elements.

```
Join(list[,delimiter])
```

Arguments

list	A one-dimensional array containing substrings to be joined.
delimiter	*Optional* — String character used to separate the substrings in the returned string. Usually a single character, but can be a multi-character 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.

Description

The **Join** function joins array elements into a string. The **Split** function does the opposite; it splits a string into array elements.

The array elements in *list* must be one-dimensional (for example, A(1), A(6), etc.). Elements are returned in ascending numeric order; elements do not have to be sequential.

Examples

The following example uses the **Join** function to join the substrings of MyArray. By default, it supplies blank spaces between elements.

```
Dim MyString, MyString2
Dim MyArray
MyArray(0) = "Mr."
MyArray(1) = "John"
MyArray(2) = "Doe"
MyArray(3) = "III"
Println Join(MyArray)   ' Returns "Mr. John Doe III".
```

The following example demonstrates the *delimiter* argument. The first **Join** function specifies a empty string; resulting in concatenated elements. The second **Join** function specifies a single-character delimiter. The third **Join** function specifies a multi-character delimiter.

```
Dim MyString, MyString2
Dim MyArray
MyArray(0) = "Mr."
MyArray(1) = "John"
MyArray(2) = "Doe"
MyArray(3) = "III"
Println Join(MyArray,"")     ' Returns "Mr.JohnDoeIII".
Println Join(MyArray,"^")    ' Returns "Mr.^John^Doe^III".
Println Join(MyArray,"^x^")  ' Returns "Mr.^x^John^x^Doe^x^III".
```

The following example demonstrate non-sequential array elements:

```
Dim MyString, MyString2
Dim MyArray
MyArray(6) = "Mr."
MyArray(4) = "John"
MyArray(3) = "Doe"
MyArray(7) = "III"
Println Join(MyArray,",")     ' Returns "Doe,John,Mr.,III".
```

See Also

- Split function

LCase

Returns a string that has been converted to lowercase.

```
LCase(string)
```

Arguments

The string argument is any valid string expression.

Description

Only uppercase letters are converted to lowercase. Lowercase letters and non-letter characters remain unchanged.

Examples

The following example uses the **LCase** function to convert uppercase letters to lowercase:

```
Dim MyString
MyString = "Caché from InterSystems"
Println LCase(MyString) ' Returns "caché from intersystems"
```

The following example converts the first four letters of the Greek alphabet from uppercase to lowercase:

```
Dim MyString
MyString = Chr(913)&Chr(914)&Chr(915)&Chr(916)
Println MyString
Println LCase(MyString)
```

(Note that the above example requires a Unicode installation of Caché.)

See Also

- UCase function

Left

Returns or replaces a specified number of characters from the left end of a string.

```
Left(string,length)
Left(string,length)=value
```

Arguments

string	String expression from which the leftmost characters are returned.
length	Numeric expression that evaluates to a positive integer indicating how many characters from the beginning of *string* to return or replace. Fractional numbers are truncated to an integer. If *length* is 0 or a negative number, **Left** returns a zero-length string (""). If *length* is 0 or a negative number, **Left** performs no replacement. If *length* is greater than or equal to the number of characters in *string*, the entire string is returned (or replaced). No padding is performed.
value	The value used to replace the specified character(s) at the beginning of string. An expression that evaluates to a string.

Description

The **Left** function can be used in two ways:

- To return a substring from the beginning (left end) of *string*. This uses the `Left(string,length)` syntax.
- To replace a substring from the beginning (left end) of *string*. The replacement substring may be the same length, longer, or shorter than the original substring. This uses the `Left(string,length)=value` syntax.

`Left(string,length)` returns the leftmost character(s) of *string*. The *length* argument specifies how many characters of *string* to return. If *length* is 0 or a negative number, **Left** returns the empty string (""). If you specify a *length* greater than the length of *string*, the entire string is returned.

`Left(string,length)=value` replaces the leftmost character(s) of *string* with *value*. The *length* argument specifies how many characters of *string* to replace. If *length* is 0 or a negative number, *string* is unchanged. This is true even when *string* is the empty string (""). If *length* is greater than the length of *string*, *string* is replaced by *value*.

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

The **Right** function returns (or replaces) the specified number of characters from the end (right end) of a string. The **Mid** function returns (or replaces) the specified number of characters from a specified starting point within a string.

Examples

The following example uses the **Left** function to return the first three characters of mystr, the first 99 characters (in this case, all of the characters), and the first 0 characters:

```
Dim mystr
mystr = "InterSystems"
Println "length 3:",Left(mystr,3)   ' Returns "Int"
Println "length 99:",Left(mystr,99) ' Returns "InterSystems"
Println "length 0:",Left(mystr,0)   ' Returns ""
```

The following example uses the **Left** function to replace the first three characters of mystr with a five-character string:

```
Dim mystr
mystr = "NtrSystems"
PrintLn mystr
Left(mystr,3)="Inter"
PrintLn mystr
```

The following example deletes (replaces with the null string) the first three characters of mystr:

```
Dim mystr
mystr = "NtrSystems"
PrintLn mystr
Left(mystr,3)=""
PrintLn mystr
```

The following example replaces all of the characters of mystr, because *length* is greater than the length of mystr:

```
Dim mystr
mystr = "Oracle"
PrintLn mystr
Left(mystr,99)="InterSystems"
PrintLn mystr
```

The following example shows that *length*=0 has no effect on mystr:

```
Dim mystr
Dim empstr
mystr = "InterSystems"
empstr = ""
PrintLn mystr
Left(mystr,0)="Bongo"
PrintLn "string out:",mystr
Left(empstr,0)="BongoSystems"
PrintLn "string out:",empstr
```

See Also

- Len function
- Mid function
- Right function

Len

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

```
Len(string | varname [,delimiter])
```

Arguments

string	Any valid string expression.
varname	Any valid variable name.
delimiter	*Optional* — A valid string expression which demarcates separate substrings in the target string. If the *delimiter* is specified but is not part of the string, the **Len** function returns 1.

Description

The **Len** function returns the number of characters in a specified string or in the value of a specified variable. Numbers are converted to canonical form, with leading and trailing zeroes and plus signs removed. An empty string ("") returns a length of 0. An undefined variable returns a length of 0.

Examples

The following example uses the **Len** function to return the number of characters in a string:

```
Println Len("InterSystems")  ' Returns 12
Println Len(+0099.900)        ' Returns 4
Println Len("0099.900")      ' Returns 8
Println Len("")              ' Returns 0
```

The following example uses the **Len** function to return the number bytes required to store a variable.

```
x = 0099.900
y = Now
Println Len(x)        ' Returns 4
Println Len(y)        ' Returns 21:
                      ' mm/dd/yyyy 00:00:00PM
Println Len(z)        ' Returns 0
```

The following example uses the **Len** function to return the number of substrings delimited by the “/” character in a string:

```
Dim MyPieces
MyPieces = Len("09/02/1994", "/") 'MyPieces contains 3
Println MyPieces," pieces of the string"
```

See Also

- InStr function
- Piece function

List

Returns elements from a list.

```
List(list[,position[,end]])
List(list[,position[,end]])=value
```

Arguments

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.
position	*Optional* — An integer that specifies the position of the list element to return, or the beginning of a sublist range if *end* is specified. Specify an expression that evaluates to a non-zero positive integer. You can use –1 to specify the last element in the list. If *position* is not specified, it defaults to 1 (the first element in *list*).
end	*Optional* — An integer that specifies the position of the list element which is the final element in the sublist range. Specify an expression that evaluates to an integer. Use –1 to specify the last element in the list.

Description

List returns elements from a list. The elements returned depend on the specified arguments.

- **List**(*list*) returns the first element string in the list.
- **List**(*list*,*position*) returns the element indicated by the specified position. The *position* argument must evaluate to an integer. List elements are numbered beginning with 1.
- **List**(*list*,*position*,*end*) returns a "sublist" containing the elements of the list from the specified start *position* through the specified *end* position, inclusive.

You can also use **ListNext** to sequentially return elements from a list.

Arguments

list

An encoded list string containing one or more elements. Lists can be created using **ListBuild** or **ListFromString**, or extracted from another list by using the **List** function. The following are valid *list* arguments:

```
myList = ListBuild("Red", "Blue", "Green", "Yellow")
Println List(myList, 2) 'prints Blue
subList = List(myList,2,4)
Println List(subList, 2) 'prints Green
```

In the following example, *subList* is not a valid *list* argument, because a **List** returns a single element as an ordinary string, not an encoded list string:

```
myList = ListBuild("Red", "Blue", "Green", "Yellow")
subList = List(myList,2)
Println List(subList,1)  ' INVALID OPERATION
```

Attempting to use the **List** function on an ordinary string generates a runtime error.

position

The position of a list element to return. List elements are counted from 1. If *position* is omitted, the first element is returned. If the value of *position* is 0 or greater than the number of elements in the list, Caché issues a <NULL VALUE> error. If the value of *position* is negative one (–1), **List** returns the final element in the list.

If the *end* argument is specified, *position* specifies the first element in a range of elements. Even when only one element is returned (when *position* and *end* are the same number) this value is returned as an encoded list string. Thus, List(x,2) is not identical to List(x,2,2), as shown in the following example:

```
MyList = ListBuild("A","B","C")
x = List(MyList,2)
y = List(MyList,2,2)
If x=y Then
  Println "Lists are identical"
Else
  Println "Lists not identical"
End If
```

end

The position of the last element in a range of elements. You must specify *position* to specify *end*. When *end* is specified, the value returned is an encoded list string. Because of this encoding, such strings should only be processed by other List functions.

If the value of *end* is:

- greater than *position*, an encoded string containing a list of elements is returned.
- equal to *position*, an encoded string containing the one element is returned.
- less than *position*, the null string ("") is returned.
- greater than the number of elements in *list*, it is equivalent to specifying the final element in the list.
- negative one (–1), it is equivalent to specifying the final element in the list.

When specifying *end*, you can specify a *position* value of zero (0). In this case, 0 is equivalent to 1.

Note that **List**(*list*,1) is not equivalent to **List**(*list*,1,1) because the former returns a string, while the latter returns a single-element list. Furthermore, the first can receive a <NULL VALUE> error, whereas the second cannot; if there are no elements to return, it returns a null string.

```
fruit = ListBuild("apple","banana","pear")
PrintLn List(fruit,1)
PrintLn List(fruit,1,1)
```

List Errors

The following **List** argument values generate an error:

- If the *list* parameter does not evaluate to a valid list, **List** generates a <LIST> error. You can use the ListValid function to determine if a list is valid.
- If the *list* parameter evaluate to a valid list that contains a null value, or concatenates a list and a null value, **List** generates a <NULL VALUE> error. All of the following are valid lists (according to ListValid) for which **List** generate a <NULL VALUE> error:

  ```
  PrintLn List("")
  PrintLn List(ListBuild())
  PrintLn LIST(ListBuild(NULL))
  PrintLn LIST(ListBuild(,))
  PrintLn LIST(ListBuild() & ListBuild("a","b","c"))
  ```

- If the *position* parameter is 0 or a fractional number less than 1 and no *end* parameter is used, **List** generates a <NULL VALUE> error.

- If the value of the *position* parameter refers to a nonexistent list member and no *end* parameter is used, **List** generates a <NULL VALUE> error.
- If the value of the *position* parameter identifies an element with an undefined value, and no *end* parameter is used, **List** generates a <NULL VALUE> error.
- If the value of the *position* parameter or the *end* parameter is less than -1, **List** generates a <RANGE> error.

Setting List

You can use **List** on the left of the equal sign to replace a specified element in a list with another element value. You can perform the following operations:

- Replace one element value with a new value:

```
fruit = ListBuild("apple","banana","pear")
PrintLn List(fruit,2)
List(fruit,2) = "orange"
PrintLn List(fruit,2)
```

- Replace a range of element values with the same number of new values:

```
fruit = ListBuild("apple","peach","pear","plum")
PrintLn List(fruit,2)," ",List(fruit,3)
List(fruit,2,3)=ListBuild("orange","banana")
PrintLn List(fruit,2)," ",List(fruit,3)
```

- Replace a range of element values with a larger or smaller number of new values:

```
fruit = ListBuild("apple","pear","plum","tangerine")
PrintLn List(fruit,2)," ",List(fruit,3)," ",List(fruit,4)
List(fruit,2,3)=ListBuild("orange","banana","peach")
PrintLn List(fruit,2)," ",List(fruit,3)," ",List(fruit,4)," ",List(fruit,5)
```

- Remove an element or a range of element values:

```
fruit = ListBuild("apple","pear","plum","tangerine")
PrintLn List(fruit,2)," ",List(fruit,3)," ",List(fruit,4)
List(fruit,2,3)=""
PrintLn List(fruit,2)
```

To replace the final element in a list use a *position* of -1. Note that `List(inlint,-1)=value` and `List(inlint,-1,-1)=value` perform different operations: `List(inlint,-1)=value` replaces the *value* of the last element; `List(inlint,-1,-1)=value` deletes the last element, then appends the specified list.

To remove the final element of a list, use `List(inlint,-1,-1)=""`.

Examples

The following examples demonstrates how to use the **List** function:

```
myList = ListBuild("Red","Blue","Green","Yellow")
color4 = List(myList,4) ' returns value of the 4th element
Println color4             ' prints Yellow
sublist = List(myList,2,3) ' returns the 2nd and 3rd elements as a list
Println List(sublist,1)    ' prints Blue
```

Because multi-element lists contain non-printing list encoding characters, **Println** should only be used to display single list items.

See Also

- ListBuild function
- ListExists function

- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListBuild

Creates a list of elements.

```
ListBuild(element[,element][,element][...])
```

Arguments

element	Any expression or comma-separated list of expressions. To include a comma within an *element*, make the *element* a quoted string.

Description

ListBuild takes one or more expressions and returns a list with one element for each expression.

Omitting an element expression yields an element whose value is undefined. For example, the following **ListBuild** statement produces a three-element list whose second element has an undefined value; referencing the second element with the **List** function will produce a Null Value error.

```
Println List(ListBuild("Red",,"Green"),2)
```

Invoking the **ListBuild** function with no arguments returns a list with one element whose data value is undefined.

An element of a list may itself be a list. For example, the following statement produces a three-element list whose third element is the two-element list, “Walnut,Pecan”:

```
MyList = ListBuild("Apple","Pear",ListBuild("Walnut","Pecan"))
Println List(MyList,3)
```

Note that multi-element lists contain non-printing list encoding characters.

The result of concatenating two lists with the Binary Concatenate operator is another list. For example, the following two **ListBuild** statements produce the same list, “A,B,C”:

```
x = ListBuild("A","B") & ListBuild("C")
y = ListBuild("A","B","C")
If x=y Then
  Println "Lists are identical"
Else
  Println "Lists not identical"
End If
```

However, concatenating a string to a list does not create a valid list.

ListBuild uses an optimized binary representation to store data elements. For this reason, equivalency tests may not work as expected with some list data. Data that might, in other contexts, be considered equivalent, may have a different internal representation. For example, **ListBuild(1)** is not equal to **ListBuild(“1”)**. This is shown in the following example:

```
x = ListBuild("1","2")
y = ListBuild(1,2)
If x=y Then
  Println "Lists are identical"
Else
  Println "Lists not identical"
End If
```

A Caché list can also be created using the **ListFromString** function, or extracted from another list using the **List** function.

Examples

The following examples demonstrates how to use the **ListBuild** function:

```
myList = ListBuild("Red","Blue","Green","Yellow")
color4 = List(myList,4) 'returns value of the 4th element
Println color4
sublist = List(myList,2,3) 'returns the 2nd and 3rd elements as a list
Println List(sublist,1) 'prints Blue
```

Because multi-element lists contain non-printing list encoding characters, **Println** should only be used to display single list items.

See Also

- List function
- ListExists function
- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListExists

Indicates whether an element is present in the list and has a value.

```
ListExists(list,position)
```

Arguments

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.
position	An integer specifying a position in *list* (counting from 1).

Description

The **ListExists** function returns a value of 1 if the element at the indicated *position* in the list exists and has a data value. Otherwise **ListExists** returns zero.

Examples

The following example demonstrates the **ListExists** function. It defines a six-element list, in which the third and fourth elements do not have a defined value:

```
Erase Y      ' Y is now undefined
myList = ListBuild("Red","Blue",,Y,"Yellow","")
Println ListExists(myList,0)  ' 0: positions are numbered from 1
Println ListExists(myList,1)  ' 1: "Red"
Println ListExists(myList,2)  ' 1: "Blue"
Println ListExists(myList,3)  ' 0: missing element
Println ListExists(myList,4)  ' 0: undefined element
Println ListExists(myList,5)  ' 1: "Yellow"
Println ListExists(myList,6)  ' 1: empty string OK
Println ListExists(myList,7)  ' 0: beyond end of list
Println ListExists(myList,-1) ' 1: last element in list
```

See Also

- List function
- ListBuild function
- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListFind

Finds an element in a list.

```
ListFind(list,value[,startafter])
```

Arguments

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.
value	An expression which evaluates to the value of the element to find.
startafter	*Optional* — An expression interpreted as a list position. The search starts with the element after this position.

Description

The **ListFind** function searches the specified *list* for the first instance of the requested *value*. The search begins with the element after the position indicated by the *startafter* argument. If you omit the *startafter* argument, **ListFind** assumes a *startafter* value of 0 and starts the search with the first element. If the value is found, **ListFind** returns the position of the first matching element. If the value is not found, **ListFind** returns zero (0). The **ListFind** function will also return a 0 if the value of the *startafter* argument refers to a nonexistent list member.

The **ListFind** function only matches complete elements. Thus, the following example returns 0 because no element of the list is equal to the string “B”, though all of the elements contain “B”:

```
mylist = ListBuild("ABC","BCD","BBB")
Println ListFind(mylist,"B")
```

Examples

The following example demonstrates how to use the **ListFind** function:

```
myList = ListBuild("Red", "Blue", "Green", "Yellow","Green")
Println ListFind(myList,"Green")   ' prints 3
Println ListFind(myList,"Green",3) ' prints 5
Println ListFind(myList,"Red")     ' prints 1
Println ListFind(myList,"Red",1)   ' prints 0 (not found)
```

See Also

- List function
- ListBuild function
- ListExists function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListFromString

Creates a list from a string.

```
ListFromString(string[,delimiter])
```

Parameters

string	A string to be converted into a Caché list. This string contains one or more elements, separated by a *delimiter*. The *delimiter* does not become part of the resulting Caché list.
delimiter	*Optional* — The delimiter used to separate substrings (elements) in *string*. Specify *delimiter* as a quoted string. If no *delimiter* is specified, the default is the comma (,) character.

Description

ListFromString takes a quoted string containing delimited elements and returns a list. A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. Lists are handled using the Caché Basic List functions.

A Caché list can also be created using the **ListBuild** function, or extracted from another list using the **List** function.

Parameters

string

A string literal (enclosed in quotation marks), a numeric, or a variable or expression that evaluates to a string. This string can contain one or more substrings (elements), separated by a *delimiter*. The string data elements must not contain the *delimiter* character (or string), because the *delimiter* character is not included in the output list.

delimiter

A character (or string of characters) used to delimit substrings within the input string. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You cannot specify a null string ("") as a delimiter.

Example

The following example takes a string containing names that are separated by blank spaces, and creates a list:

```
namestring="Deborah Noah Martha Bowie"
namelist=ListFromString(namestring," ")
Println "1st element: ",List(namelist,1)
Println "2nd element: ",List(namelist,2)
Println "3rd element: ",List(namelist,3)
```

The blank spaces are the string *delimiter* and are not included in the newly created list.

The following example takes a string containing names that are separated by a <sp> delimiter, and creates a list:

```
namestring="Deborah<sp>Noah<sp>Martha<sp>Bowie"
namelist=ListFromString(namestring,"<sp>")
Println "1st element: ",List(namelist,1)
Println "2nd element: ",List(namelist,2)
Println "3rd element: ",List(namelist,3)
```

The following example uses the default delimiter (a comma) and creates a list. Note that the second element of this list contains no value:

```
namestring="Deborah,,Noah,Martha"
namelist=ListFromString(namestring)
Println "1st element: ",List(namelist,1)
Println "2nd element: ",List(namelist,2)
Println "3rd element: ",List(namelist,3)
```

See Also

- List function
- ListBuild function
- ListExists function
- ListFind function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function
- Piece function

ListGet

Returns an element from a list.

```
ListGet(list[,position[,default]])
```

Arguments

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.
position	*Optional* — An integer specifying the position of the element (counting from 1).
default	*Optional* — An expression that provides the value to return if the list element has an undefined value.

Description

ListGet returns the requested element in the specified list. If the value of *position* refers to a nonexistent member or identifies an element with an undefined value, the specified *default* value is returned.

The **ListGet** function is identical to the one- and two-argument forms of the **List** function except that, under conditions that would cause **List** to produce a <NULL VALUE> error, **ListGet** returns a default value. See the description of the **List** function for more information on conditions that return <NULL VALUE> errors.

Arguments

position

An integer that specifies the target list element. If it is omitted, the function defaults to examine the first element of the list. If the value of *position* is -1, **ListGet** examines the last element of the list.

default

Supplies a default value to return when the element has no value. For example, an omitted element (1,,3) or an undefined variable (1,x,3). The *default* is not returned for an empty (zero-length) string value. If you omit the *default* argument, a zero-length string is assumed for the default value.

Examples

The following example demonstrates how to use the **ListGet** function:

```
myList = ListBuild("Red","Blue",,"Yellow")
Println ListGet(myList)           ' prints Red
Println ListGet(myList,2,"White") ' prints Blue
Println ListGet(myList,3,"White") ' prints White
```

The following example shows the difference between an undefined value and an empty string value:

```
myList = ListBuild("Red","",,"Yellow")
Println "Empty: ",ListGet(myList,2,"White")   ' prints empty string
Println "Default: ",ListGet(myList,3,"White") ' prints White
```

See Also

- List function
- ListBuild function
- ListExists function

- ListFind function
- ListFromString function
- ListLength function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListLength

Returns the number of elements in a list.

```
ListLength(list)
```

Arguments

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. The null string ("") is also treated as a valid list.

Description

ListLength returns the number of elements in *list*. It counts all elements in a list, regardless of whether the element has a data value.

Examples

The following example demonstrates how to use the **ListLength** function:

```
myList = ListBuild("Red","Blue","Green","Yellow")
Println ListLength(myList) 'prints 4
```

The following example shows that **ListLength** counts all list elements:

```
GapList = ListBuild("Red",, "Green", "Yellow")
UndefVarList = ListBuild("Red",x, "Green", "Yellow")
NullStrList = ListBuild("Red","", "Green", "Yellow")
Println ListLength(GapList)        ' prints 4
Println ListLength(UndefVarList)   ' prints 4
Println ListLength(NullStrList)    ' prints 4
```

The following example shows how **ListLength** handles the null string and a list containing only a null string element:

```
Println ListLength("")       '  prints 0
NullList = ListBuild("")
Println ListLength(NullList) '  prints 1
```

See Also

- List function
- ListBuild function
- ListExists function
- ListFind function
- ListFromString function
- ListGet function
- ListNext function
- ListSame function
- ListToString function
- ListValid function

ListNext

Retrieves elements sequentially from a list.

```
ListNext(list,ptr,value)
```

Parameters

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.
ptr	A pointer to the next element in the list. You must specify *ptr* as a local variable initialized to 0 to point to the beginning of *list*. Caché increments *ptr* using an internal address value algorithm (*not* a predictable integer counter). Therefore, the only value you can use to set *ptr* is 0. *ptr* cannot be a global variable or a subscripted variable.
value	A local variable used to hold the data value of a list element. *value* does not have to be initialized before invoking **ListNext**. *value* cannot be a global variable or a subscripted variable.

Description

ListNext sequentially returns elements from *list*. You initialize *ptr* to 0 before the first invocation of **ListNext**. This causes **ListNext** to begin returning elements from the beginning of the list. Each successive invocation of **ListNext** advances *ptr* and returns the next list element value to *value*. The **ListNext** function returns 1, indicating that a list element has been successfully retrieved.

When **ListNext** reaches the end of the list, it returns 0, resets *ptr* to 0, and leaves *value* unchanged from the previous invocation. Because *ptr* has been reset to 0, the next invocation of **ListNext** would start at the beginning of the list.

Caché Basic increments *ptr* using an internal address algorithm. Therefore, the only value you should use to set *ptr* is 0.

You can use **ListValid** to determine if *list* is a valid list. An invalid list causes **ListNext** to generate a <LIST> error.

Not all lists validated by **ListValid** can be used successfully with **ListNext**. When **ListNext** encounters a list element with a null value, it returns 1 indicating that a list element has been successfully retrieved, advances *ptr* to the next element, and resets *value* to be an undefined variable. This can happen with any of the following valid lists: *value*=ListBuild(), *value*=ListBuild(NULL), *value*=ListBuild(,), or when encountering an omitted list element, such as the second invocation of **ListNext** on *value*=ListBuild("a",,"b").

`ListNext("",ptr,value)` returns 0, and does not advance the pointer or set *value*.
`ListNext(ListBuild(""),ptr,value)` returns 1, advances the pointer, and set *value* to the null string ("").

ListNext and Nested Lists

The following example returns three elements, because **ListNext** does not recognize the individual elements in nested lists:

```
mylist = ListBuild("Apple","Pear",ListBuild("Walnut","Pecan"))
ptr = 0
count = 0
   While ListNext(mylist,ptr,value)
    count=count+1
    PrintLn value
   Wend
 PrintLn "End of list: ",count," elements found"
```

Examples

The following example sequentially returns all the elements in the list:

```
mylist = ListBuild("Red","Blue","Green")
ptr = 0
count = 0
   While ListNext(mylist,ptr,value)
    count = count+1
    PrintLn value
   Wend
   PrintLn "End of list: ",count," elements found"
```

See Also

- List function
- ListBuild function
- ListExists function
- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListSame function
- ListToString function
- ListValid function

ListSame

Compares two lists and returns a boolean value.

```
ListSame(list1,list2)
```

Parameters

list1	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. The null string ("") is also treated as a valid list.
list2	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. The null string ("") is also treated as a valid list.

Description

ListSame compares the contents of two lists and returns 1 if the lists are identical. If the lists are not identical, **ListSame** returns 0. **ListSame** compares list elements using their string representations. **ListSame** comparisons are case-sensitive.

ListSame compares the two lists element-by-element in left-to-right order. Therefore **ListSame** returns a value of 0 when it encounters the first non-identical pair of list elements; it does not check subsequent items to determine if they are valid list elements. If a **ListSame** comparison encounters an invalid item, it issues a <LIST> error.

Examples

The following example returns 1, because the two lists are identical:

```
x = ListBuild("Red","Blue","Green")
y = ListBuild("Red","Blue","Green")
 PrintLn ListSame(x,y)
```

The following example returns 0, because the two lists are not identical:

```
 x = ListBuild("Red","Blue","Yellow")
 y = ListBuild("Red","Yellow","Blue")
 PrintLn ListSame(x,y)
```

Identical Lists

ListSame considers two lists to be identical if the string representations of the two lists are identical.

When comparing a numeric list element and a string list element, the string list element must represent the numeric in canonical form; this is because Caché always reduces numerics to canonical form before performing a comparison. In the following example, **ListSame** compares a string and a numeric. The first three **ListSame** functions return 1 (identical); the fourth **ListSame** function returns 0 (not identical) because the string representation is not in canonical form:

```
PrintLn ListSame(ListBuild("365"),ListBuild(365))
PrintLn ListSame(ListBuild("365"),ListBuild(365.0))
PrintLn ListSame(ListBuild("365.5"),ListBuild(365.5))
PrintLn ListSame(ListBuild("365.0"),ListBuild(365.0))
```

ListSame comparison is not the same equivalence test as the one used by other list operations, which test using the internal representation of a list. This distinction is easily seen when comparing a number and a numeric string, as in the following example:

```
x = ListBuild("365")
y = ListBuild(365)
  If x=y Then
     PrintLn "Equal sign: number/numeric string identical"
  Else
     PrintLn "Equal sign: number/numeric string differ"
  End If
  If 1=ListSame(x,y) Then
     PrintLn "ListSame: number/numeric string identical"
  Else
     PrintLn "ListSame: number/numeric string differ"
  End If
```

The equality (=) comparison tests the internal representations of these lists (which are not identical). **ListSame** performs a string conversion on both lists, compares them, and finds them identical.

The following example shows two lists with various representations of numeric elements. **ListSame** considers these two lists to be identical:

```
x = ListBuild("360","361","362","363","364","365","366")
y = ListBuild(00360.000,(19*19),+"362",363,364.0,+365,"3" & "66")
PrintLn ListSame(x,y)," lists are identical"
```

Null String and Null List

A list containing the null string (an empty string) as its sole element is a valid list. The null string by itself is also considered a valid list. However these two (a null string and a null list) are not considered identical, as shown in the following example:

```
PrintLn ListSame(ListBuild(""),ListBuild(""))," null lists"
PrintLn ListSame("","")," null strings"
PrintLn ListSame(ListBuild(""),"")," null list and null string"
```

Normally, a string is not a valid **ListSame** argument, and **ListSame** issues a <LIST> error. However, the following **ListSame** comparisons complete successfully and return 0 (values not identical). The null string and the string “abc” are compared and found not to be identical. These null string comparisons do not issue a <LIST> error:

```
PrintLn ListSame("","abc")
PrintLn ListSame("abc","")
```

The following **ListSame** comparisons do issue a <LIST> error, because a list (even a null list) cannot be compared with a string:

```
x = ListBuild("")
PrintLn ListSame("abc",x)
PrintLn ListSame(x,"abc")
```

Comparing “Empty” Lists

ListValid considers all of the following as valid lists:

```
PrintLn ListValid("")
PrintLn ListValid(ListBuild())
PrintLn ListValid(ListBuild(NULL))
PrintLn ListValid(ListBuild(""))
PrintLn ListValid(ListBuild(Chr(0)))
PrintLn ListValid(ListBuild(,))
```

ListSame considers only the following pairs as identical:

```
PrintLn ListSame(ListBuild(),ListBuild(NULL))
PrintLn ListSame(ListBuild(,),ListBuild(NULL,NULL))
PrintLn ListSame(ListBuild(,),ListBuild() & ListBuild())
```

Empty Elements

A **ListBuild** can create empty elements by including extra commas between elements or appending one or more commas to either end of the element list. **ListSame** is aware of empty elements, and does not treat them as equivalent to null string elements.

The following **ListSame** examples all return 0 (not identical):

```
PrintLn ListSame(ListBuild(365,,367),ListBuild(365,367))
PrintLn ListSame(ListBuild(365,366,),ListBuild(365,366))
PrintLn ListSame(ListBuild(365,366,,),ListBuild(365,366,))
PrintLn ListSame(ListBuild(365,,367),ListBuild(365,"",367))
```

Nested and Concatenated Lists

ListSame does not support nested lists. It cannot compare two lists that contain lists, even if their contents are identical.

```
 x = ListBuild("365")
 y = ListBuild(365)
 PrintLn ListSame(x,y)," lists identical"
 PrintLn ListSame(ListBuild(x),ListBuild(y))," nested lists not identical"
```

In the following example, both **ListSame** comparisons returns 0, because these lists are not considered identical:

```
 x=ListBuild("Apple","Pear","Walnut","Pecan")
 y=ListBuild("Apple","Pear",ListBuild("Walnut","Pecan"))
 z=ListBuild("Apple","Pear","Walnut","Pecan","")
 PrintLn ListSame(x,y)," nested list"
 PrintLn ListSame(x,z)," null string is list item"
```

ListSame does support concatenated lists. The following example returns 1, because the lists are considered identical:

```
 x=ListBuild("Apple","Pear","Walnut","Pecan")
 y=ListBuild("Apple","Pear") & ListBuild("Walnut","Pecan")
 PrintLn ListSame(x,y)," concatenated list"
```

See Also

- List function
- ListBuild function
- ListExists function
- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListToString function
- ListValid function

ListToString

Creates a string from a list.

```
ListToString(list[,delimiter])
```

Parameters

list	An expression that evaluates to a valid list. A Caché *list* must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. The null string ("") is also treated as a valid list.
delimiter	*Optional* — A delimiter used to separate substrings. Specify *delimiter* as a quoted string. If no *delimiter* is specified, the default is the comma (,) character.

Description

ListToString takes a Caché list and converts it to a string. In the resulting string, the elements of the list are separated by the *delimiter*.

A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. **ListToString** converts this list to a string with delimited elements. It sets aside a specified character (or character string) to serve as a delimiter. These delimited elements can be handled using the **Piece** function.

Note: The *delimiter* specified here must not occur in the source data. Caché makes no distinction between a character serving as a delimiter and the same character as a data character.

Parameters

list

A Caché list, which contains one or more elements. A list is created using **ListBuild** or **ListFromString**, or extracted from another list using **List**.

delimiter

A character (or string of characters) used to delimit substrings within the output string. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You can specify a null string ("") as a delimiter; in this case, substrings are concatenated with no delimiter. To specify a quote character as a delimiter, specify the quote character twice (""""") or use Char(34).

Examples

The following example creates a list of four elements, then converts it to a string with the elements delimited by the colon (:) character:

```
namelist=ListBuild("Deborah","Noah","Martha","Bowie")
PrintLn ListToString(namelist,":")
```

returns `"Deborah:Noah:Martha:Bowie"`

The following example creates a list of four elements, then converts it to a string with the elements delimited by the *sp* string:

```
namelist=ListBuild("Deborah","Noah","Martha","Bowie")
PrintLn ListToString(namelist,"*sp*")
```

returns `"Deborah*sp*Noah*sp*Martha*sp*Bowie"`

See Also

- List function
- ListBuild function
- ListExists function
- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListValid function
- Piece function

ListValid

Determines if an expression is a list.

```
ListValid(exp)
```

Parameters

exp	Any valid expression. A valid list must be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. The null string ("") is also treated as a valid list.

Description

ListValid determines whether *exp* is a list, and returns a Boolean value: If *exp* is a list, **ListValid** returns 1; if *exp* is not a list, **ListValid** returns 0.

A list can be created using **ListBuild** or **ListFromString**, or extracted from another list using **List**. A list containing the empty string ("") as its sole element is a valid list. The empty string ("") by itself is also considered a valid list.

Examples

The following examples all return 1, indicating a valid list:

```
w = ListBuild("Red","Blue","Green")
x = ListBuild("Red")
y = ListBuild(365)
z = ListBuild("")
Println ListValid(w)
Println ListValid(x)
Println ListValid(y)
Println ListValid(z)
```

The following examples all return 0. Numbers and strings (with the exception of the null string) are not valid lists:

```
x = "Red"
y = 44
Println ListValid(x)
Println ListValid(y)
```

The following examples all return 1. Concatenated, nested, and omitted value lists are all valid lists:

```
w = ListBuild("Apple","Pear")
x = ListBuild("Walnut","Pecan")
y = ListBuild("Apple","Pear",ListBuild("Walnut","Pecan"))
z = ListBuild("Apple","Pear",,"Pecan")
Println ListValid(w & x)    ' concatenated
Println ListValid(y)      ' nested
Println ListValid(z)      ' omitted element
```

The following examples all return 1. **ListValid** considers all of the following “empty” lists as valid lists:

```
Println ListValid("")
Println ListValid(ListBuild())
Println ListValid(ListBuild(NULL))
Println ListValid(ListBuild(""))
Println ListValid(ListBuild(Chr(0)))
Println ListValid(ListBuild(,))
```

See Also

- List function
- ListBuild function
- ListExists function

- ListFind function
- ListFromString function
- ListGet function
- ListLength function
- ListNext function
- ListSame function
- ListToString function

Lock

Obtains a logical lock on a variable name.

```
Lock(varname[,timeout])
```

Arguments

varname	Name of the variable to be locked.
timeout	*Optional* — A numeric expression indicating the number of seconds to wait to obtain the lock.

Description

Returns true if the lock was obtained, false otherwise.

Each time a lock is obtained on a *varname* a lock count is incremented for this *varname*. **Unlock** decrements this count. Only when the lock count falls to zero will the logical lock be released. For this reason, you should balance each call to **Lock** with a corresponding call to **Unlock**.

If a *timeout* is not specified, **Lock** will wait indefinitely for the lock to be obtained. Specifying a *timeout* causes the lock attempt to wait up to the timeout number of seconds to obtain the lock.

Example

The following example uses the **Lock** function to obtain a logical lock on a *varname* with a timeout of 10 seconds.

```
If Lock(^PatientData(PatientID), 10) = True Then
  Println "Got the Lock"
  Unlock(^PatientData(PatientID))
Else
  Println "Couldn't get the lock"
End If
```

See Also

- Unlock function

Log

Returns the natural logarithm of a number.

```
Log(number)
```

Arguments

The number argument can be any valid numeric expression greater than 0. Specifying 0 or a negative number results in a runtime error.

Description

The natural logarithm is the logarithm to the base e. The constant e (**Exp(1)**) 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:

```
Logn(x) = Log(x) / Log(n)
```

Note: In ObjectScript, the equivalent function is the $ZLN function, which returns the natural logarithm. The $ZLOG function returns the base-10 logarithm. Please note that **Log** and **$ZLOG** are *not* equivalent functions.

Examples

The following example uses the **Log** function to calculate the natural logarithm for each of the integers 1 through 10:

```
For x = 1 To 10
Println "Natural log of ",x," = ",Log(x)
Next
```

The following example uses the **Log** function to calculate the base-10 logarithms for the numbers 10 through 100, counting by tens. For 100, it returns 2 (10 to the power of *x* = 100).

```
x = 10
For y = 1 To 10
Base10 = Log(x) / Log(10)
Println "Base-10 log of ",x," = ",Base10
x = x + 10
Next
```

See Also

- Exp function
- Derived Math Functions

Mid

Returns or replaces a specified number of characters from a string.

```
Mid(string,start[,length])
Mid(string,start[,length])=value
```

Arguments

string	String expression from which characters are returned.
start	A positive integer specifying the character position in *string* (counting from 1) at which the substring begins.
length	*Optional* — A positive integer specifying the number of characters to return (or replace) from the *start* location (inclusive). If *length* is omitted, all characters from the *start* position to the end of the string are returned (or replaced).
value	The value used to replace the specified character(s) in *string*. An expression that evaluates to a string.

Description

The **Mid** function can be used in two ways:

- To return a substring from *string*. The substring is determined by specifying the *start* position and, optionally, the *length*. This uses the `Mid(string,start,length)` syntax.
- To replace a substring within *string*. The replacement substring may be the same length, longer, or shorter than the original substring. The substring is determined by specifying the *start* position and, optionally, the *length*. This uses the `Mid(string,start,length)=value` syntax.

`Mid(string,start,length)` returns the character(s) of *string* specified by the *start* and *length* arguments. The *length* argument specifies how many characters of *string* to return. If *length* is 0 or a negative number, **Mid** returns the empty string (""). If you specify a *length* greater than the number of available characters, all of the characters to the right of *start* are returned.

`Mid(string,start,length)=value` replaces the specified character(s) of *string* with *value*.

The *start* argument specifies where to begin the replacement. If *start* is 1, begin at the beginning of *string*. If *start* is greater than the length of *string*, the *string* is padded with blank spaces until *start* is reached, then *value* is appended. If *start* is 0 or a negative number, the number of characters replaced is *start* plus *length* minus 1.

The optional *length* argument specifies how many characters of *string* to replace. If *length* is omitted, all of the characters to the right of *start* are replaced. If *length* is 0 or a negative number, *string* is unchanged. This is true even when *string* is the empty string ("").

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

Note that the *length* argument refers to the source length, not the replacement length, which may be longer or shorter than the substring replaced.

You can perform similar substring return and replace operations in ObjectScript using the **$EXTRACT** function.

Examples

The following example returns a substring without specifying a *length*. It begins with the twelfth character (inclusive) and returns the rest of the string:

```
Dim MyVar
MyVar = Mid("Caché is a powerful database!",12)
Println MyVar   ' Returns "powerful database!"
```

The following example returns a substring specifying a *length*. It begins with the twelfth character in a string, and returns eight characters:

```
Dim MyVar
MyVar = Mid("Caché is a powerful database!",12,8)
Println MyVar     ' Returns "powerful"
```

In the following example, all of the **Mid** functions return the empty string:

```
Dim MyVar
MyVar = Mid("Caché is a powerful database!",0,8)
Println "0,n=",MyVar
MyVar = Mid("Caché is a powerful database!",8,0)
Println "n,0=",MyVar
MyVar = Mid("Caché is a powerful database!",8,-1)
Println "n,-1=",MyVar
```

The following example show the difference between specifying a numeric as the string argument, and specifying the same value as a string. Numerics are converted to canonical form, which in this case means that leading zeros are omitted.

```
Println Mid(00123456,3,2)    ' Returns 34
Println Mid("00123456",3,2)  ' Returns 12
```

Replacement Examples

The following example replaces characters starting with specified *start* location to the end of the string:

```
var1="ABCD"
var2="ABCD"
var3="ABCD"
var4="ABCD"
var5="ABCD"
PrintLn var
Mid(var1,2)="Z"
PrintLn "start 2: ",var1
Mid(var2,8)="Z"
PrintLn "start 8: ",var2
Mid(var3,1)="Z"
PrintLn "start 1: ",var3
Mid(var4,0)="Z"
PrintLn "start 0: ",var4
Mid(var5,-1)="Z"
PrintLn "start -1: ",var5
```

The following example starts at various specified *start* locations and replaces *length*=2 characters:

```
var1="ABCD"
var2="ABCD"
var3="ABCD"
Mid(var1,2,2)="xyz"
PrintLn "start 2: ",var1
Mid(var2,8,2)="xyz"
PrintLn "start 8: ",var2
Mid(var3,1,2)="xyz"
PrintLn "start 1: ",var3
```

The following example starts at *start*=2 and replaces various *length* values:

```
var1="ABCD"
var2="ABCD"
var3="ABCD"
var4="ABCD"
var5="ABCD"
Mid(var1,2,2)="xyz"
PrintLn "length 2: ",var1
Mid(var2,2,8)="xyz"
PrintLn "length 8: ",var2
Mid(var3,2,1)="xyz"
PrintLn "length 1: ",var3
Mid(var4,2,0)="xyz"
PrintLn "length 0: ",var4
Mid(var5,2,-1)="xyz"
PrintLn "length -1: ",var5
```

The following example demonstrates replacement with *start* locations less than 1. The formula is *start*+*length*-1:

```
var1="ABCD"
var2="ABCD"
var3="ABCD"
Mid(var1,-2,5)="xyz"
PrintLn "start -2 for 5: ",var1
Mid(var2,0,1)="xyz"
PrintLn "start 0 for 1: ",var2
Mid(var3,0,2)="xyz"
PrintLn "start 0 for 2: ",var3
```

See Also

- Left function
- Len function
- LTrim, RTrim and Trim functions
- Right function

Minute

Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

```
Minute(time)
```

Arguments

The *time* argument is any expression that can represent a time. This includes a time value such as “12:30” or “1:27:55”, a time/date value such as “11/12/1999 12:33:00”, or a date value such as “11/12/1999”. If only a date is specified, the time defaults to 00:00:00. Fractional seconds are permitted, but ignored; they are truncated, not rounded.

Examples

The following example uses the **Minute** function to obtain the minute from the current time:

```
Dim MyTime, MyMin
MyTime = Now
MyMin = Minute(MyTime)
Println MyMin
```

The following example returns a minute value of 59:

```
Dim MyMin
MyMin = Minute("13:59:59.999")
Println MyMin
```

See Also

- Day function
- Hour function
- Now function
- Second function
- Time function

Month

Returns the month of the year as an integer between 1 and 12, inclusive.

```
Month(date)
```

Arguments

The *date* argument is any expression that represents a date as a string.

Description

The **Month** function locates and returns the month portion of a date string as an integer. It performs no range validation on this number. The **Month** function accepts blanks, slashes (/), hyphens (-), or commas (,) (in any combination) as date component separators. Leading zeros and plus and minus signs may be included or omitted in the input string; leading zeros and signs are omitted from the output integer. The **Month** function locates the month portion in one of the following three ways:

- In American numeric format, the month precedes the day. For example, “9/27/2005” or “9–27.” In this format, the **Month** function identifies the month portion by position. It accepts any non-numeric character as a date component separator and returns the number that precedes this first non-numeric character. It does not parse the day or year components of the date string. However, there must be at least one non-numeric character following the number specifying the month. The day and year can be any alphanumeric value, and can include or omit punctuation characters such as periods or apostrophes. The year component may be 4-digits, less than 4 digits, or omitted. If the **Month** function is unable to identify the month portion of the string, it returns 0.
- In American written format, the name of the month precedes the day. For example, “September 27 2005” or “Sept 27” In this case, the month name is validated; the first three letters must correspond to a valid month name. Validation is not case-sensitive. The month name must be followed by a valid date component separator; it cannot be followed by a period; thus “Sep” or “Sept”, but not “Sept.” If the **Month** function is unable to identify the month portion of the string, it returns 0.
- In European written format, the day precedes the name of the month. For example, “27 September 2005” or “27 Sept” In this case, the month name is validated; the first three letters must correspond to a valid month name. Validation is not case-sensitive. If the **Month** function is unable to identify the month portion of the string, it returns the day portion of the string.

Examples

The following example uses the **Month** function to return the current month:

```
Dim MonthNum
MonthNum = Month(Now)
Print MonthNum
```

The following examples uses the **Month** function to return the month from the specified date:

```
Dim MyMonth
MyMonth = Month("09/19/05")  'MyMonth contains 9.
Print MyMonth

Dim MyMonth
MyMonth = Month("Sept 19, 2005")  'MyMonth contains 9.
Print MyMonth

Dim MyMonth
MyMonth = Month("19 October 2005")  'MyMonth contains 10.
Print MyMonth
```

```
Dim MyMonth
MyMonth = Month("19 Feb")  'MyMonth contains 2.
Print MyMonth
```

See Also

- Basic: Date function, Day function, Hour function, Minute function, Now function, Second function, Weekday function, Year function
- ObjectScript: $ZDATE function
- SQL: MONTH function

MonthName

Returns a string indicating the specified month.

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

Arguments

month	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 (0), which means that the month name is not abbreviated.

Examples

The following example uses the **MonthName** and **Month** functions to return the month name for a date expression:

```
Dim MName
Mydate = "1/12/1953"
MName = MonthName(Month(Mydate))
Println MName
```

The following example uses the **MonthName** and **Date** functions to return the abbreviated month name for the current date:

```
Dim MName
CurrDate = Date
MNum = Month(CurrDate)
MName = MonthName(MNum,1)
Println MName
```

In the following example, a month number is out of the range of 1 through 12. **MonthName** returns a question mark:

```
Dim MyVar
MyVar = MonthName(13)
Println MyVar
```

See Also

- Month function
- WeekdayName function

Now

Returns the current date and time according to the setting of your computer's system date and time.

```
Now
```

Arguments

none

Description

The **Now** function returns the current date and time in a format such as the following:

```
mm/dd/yyyy 00:00:00PM
```

The exact display format depends on your system configuration. Leading zeros are displayed. The year is displayed as four digits.

To return just the current date, use the **Date** function. To return just the current time, use the **Time** function.

Examples

The following example uses the **Now** function to return the current date and time:

```
Dim MyVar
MyVar = Now
Println MyVar
```

See Also

- Basic: Sleep statement, Date function, Time function
- ObjectScript: $HOROLOG special variable
- SQL: NOW function

Oct

Returns a string representing the octal value of a number.

```
Oct(number)
```

Arguments

The *number* argument is any valid expression that resolves to a positive or negative number. If *number* is a decimal fraction, it is truncated to a whole number before being evaluated.

Description

The **Oct** function converts a number from decimal (base 10) to octal (base 8). To convert a number from octal to decimal, represent octal numbers directly by preceding numbers in the proper range with &O For example, &O10 is the octal notation for decimal 8.

If Number Is	Oct Returns
Empty	Zero (0).
Any other number	Up to 11 octal characters.

Examples

The following example uses the **Oct** function to return the octal value of a decimal (base-10) number:

```
Println Oct(4)     ' Returns 4.
Println Oct(8)     ' Returns 10.
Println Oct(459)   ' Returns 713.
```

The following example uses the &O prefix to return the decimal (base-10) value for an octal number:

```
Println &O4     ' Returns 4.
Println &O10    ' Returns 8.
Println &O713   ' Returns 459.
```

See Also

- Hex function

Piece

Returns the specified substring, using a delimiter.

```
Piece(string,delimiter[,from[,to]])
Piece(string,delimiter[,from[,to]])=value
```

Arguments

string	String expression containing substrings to be extracted.
delimiter	A delimiter used to identify substrings within *string*.
from	*Optional* — An integer that specifies the substring, or the beginning of a range of substrings, to return from the target string. Specified as a substring count from the beginning of *string*. Substrings are separated by a *delimiter*, and counted from 1. If omitted, the first substring is returned.
to	*Optional* — An integer that specifies the ending substring for a range of substrings to return from the target string. Specified as a substring count from the beginning of *string*. Must be used with *from*.

Description

The **Piece** function can be used in two ways:

- To return a substring or a range of substrings from *string*. The substring is determined by specifying a *delimiter* character (or character string) that is found in *string*. This uses the `Piece(string,delimiter,from,to)` syntax.
- To replace a substring within *string*. The replacement substring may be the same length, longer, or shorter than the original substring. The substring is determined by specifying a *delimiter* character (or character string) that is found in *string*. This uses the `Piece(string,delimiter,from,to)=value` syntax.

When returning a specified substring (piece) from *string*, the substring returned depends on the arguments used:

- **Piece**(*string*,*delimiter*) returns the first substring in *string*. If *delimiter* occurs in *string*, this is the substring that precedes the first occurrence of *delimiter*. If *delimiter* does not occur in *string*, the returned substring is *string*.
- **Piece**(*string*,*delimiter*,*from*) returns the substring which is the *n*th piece of *string*, where the integer *n* is specified by the *from* argument, and pieces are separated by a *delimiter*. The delimiter is not returned.
- **Piece**(*string*,*delimiter*,*from*,*to*) returns a range of substrings including the substring specified in *from* through the substring specified in *to*. This four-argument form of **Piece** returns a string that includes any intermediate occurrences of *delimiter* that occur between the *from* and *to* substrings. If *to* is greater than the number of substrings, the returned substring includes all substrings to the end of *string*.

The values for *from* and *to* must be positive integers when specified, otherwise the **Piece** function will return an empty string.

Arguments

string

The target string from which the substring is to be returned. It can be a string literal, a variable name, or any valid Caché Basic expression that evaluates to a string. If you specify a null string ("") as the target string, **Piece** returns the null string.

A target string usually contains instances of a character (or character string) which are used as delimiters. This character or string cannot also be used as a data value within *string*.

When **Piece** is used on the left hand side of the equals sign, *string* must evaluate to a valid variable name.

delimiter

The search string to be used to delimit substrings within *string*. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character search string, the individual characters of which can be used within string data.

If the specified delimiter is not in *string*, **Piece** returns the entire the *string* string. If the specified delimiter is the null string (""), **Piece** returns the null string.

from

The number of a substring within *string*, counting from 1. It must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. Substrings are separated by delimiters.

- If the *from* argument is omitted or set to 1, **Piece** returns the first substring of *string*. If *string* does not contain the specified delimiter, a *from* value of 1 returns *string*.
- If the *from* argument identifies by count the last substring in *string*, this substring is returned, regardless of whether it is followed by a delimiter.
- If the value of *from* is the null string (""), zero, a negative number, or greater than the number of substrings in *string*, **Piece** returns a null string.

If the *from* argument is used with the *to* argument, it identifies the start of a range of substrings to be returned as a string, and should be less than the value of *to*.

to

The number of the substring within *string* that ends the range initiated by the *from* argument. The returned string includes both the *from* and *to* substrings, as well as any intermediate substrings and the delimiters separating them. The *to* argument must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. The *to* argument must be used with *from* and should be greater than the value of *from*.

- If *from* is less than *to*, **Piece** returns a string consisting of all of the delimited substrings within this range, including the *from* and *to* substrings. This returned string contains the substrings and the delimiters within this range.
- If *to* is greater than the number of delimited substrings, the returned string contains all the string data (substrings and delimiters) beginning with the *from* substring and continuing to the end of the *string* string.
- If *from* is equal to *to*, the *from* substring is returned.
- If *from* is greater than *to*, **Piece** returns a null string.
- If *to* is the null string (""), **Piece** returns a null string.

Replacing a Substring Using Piece

You can use **Piece** to the left of the equals sign to replace a substring within *string*. When used to the left of the equals sign, **Piece** designates a substring to be replaced by the assigned value.

The use of **Piece** (and List) in this context differs from other standard functions because it modifies an existing value, instead of just returning a value.

Replacing a Delimited Substring

The following example changes the value of *colorlist* to "Red,Cyan,Yellow,Green,Orange,Purple,Black":

```
colorlist="Red,Blue,Yellow,Green,Orange,Purple,Black"
PrintLn colorlist
Piece(colorlist,",",2)="Cyan"
PrintLn colorlist
```

The replacement substring may, of course, be longer or shorter than the original.

If you do not specify a *from* argument, the first substring is replaced:

```
colorlist="Red,Blue,Yellow,Green,Orange,Purple,Black"
PrintLn colorlist
Piece(colorlist,",")="Crimson"
PrintLn colorlist
```

If you specify a *from* and *to* argument, the included substrings are replaced by the specified value, in this case the 4th, 5th, and 6th delimited substrings:

```
colorlist="Red,Blue,Yellow,Green,Orange,Purple,Black"
PrintLn colorlist
Piece(colorlist,",",4,6)="non-primary colors"
PrintLn colorlist
```

If **Piece** specifies more occurrences of the delimiter than exist in the target variable, it appends additional delimiters to the end of the value, up to one less than the specified number. The following example changes the value of *smallcolor* to "Green;Blue;;Red". The number of delimiter characters added is equal to the *from* value, minus the number of existing delimiters, minus one:

```
smallcolor="Green;Blue"
PrintLn smallcolor
Piece(smallcolor,";",4)="Red"
PrintLn smallcolor
```

If *delimiter* doesn't appear in *string*, **Piece** treats *string* as a single piece and performs the same substitutions described above. If there is no *from* argument specified, the new value replaces the original *string*:

```
colorlist="Red,Green,Blue"
PrintLn colorlist
Piece(colorlist,"^")="Purple^Orange"
PrintLn colorlist
```

If *delimiter* doesn't appear in *string*, and *from* is specified, **Piece** may append delimiters to the end of *string* and append the new value to string, to fulfill the *from* value:

```
colorlist="Red,Green,Blue"
PrintLn colorlist
Piece(colorlist,"^",3)="Purple^Orange"
PrintLn colorlist
```

Delimiter is Null String

If the delimiter is the null string, the new value replaces the original *string*, regardless of the values of the *from* and *to* arguments.

The following two examples both set *colorlist* to “Purple”:

```
colorlist="Red,Green,Blue"
PrintLn colorlist
Piece(colorlist,"")="Purple"
PrintLn colorlist

colorlist="Red,Green,Blue"
PrintLn colorlist
Piece(colorlist,"",3,5)="Purple"
PrintLn colorlist
```

Initializing a String Variable

The *string* variable does not need to be defined before being assigned a value. The following example initializes *newvar* to the character pattern ">>>>>>TOTAL":

```
Piece(newvar,">",7)="TOTAL"
PrintLn newvar
```

Piece with Parameters over 32,768 Characters

If you wish to use **Piece** with a parameter greater than 32767 characters, long strings must be enabled. Long string support is enabled by default. In the Management Portal navigate to **[System] > [Configuration] > [Memory and Startup]** from the **System Administration > Configuration > System Configuration** menu. To enable support for long strings system-wide, select the **Enable Long Strings** check box. Then click the Save button.

Examples

The following example returns designated substrings from a string using the "|" character as a delimiter:

```
MyString = "InterSystems|One Memorial Drive|Cambridge|MA 02142"
Println Piece(MyString, "|") 'InterSystems
Println Piece(MyString, "|", 2) 'One Memorial Drive
Println Piece(MyString, "|", 3, 4) 'Cambridge|MA 02142
Println Piece(Piece(MyString, "|", 4), " ") 'MA
```

The following example performs the same operation using a multicharacter delimiter string:

```
MyString = "InterSystemslinebreakOne Memorial DrivelinebreakCambridge MAlinebreak02142"
Println Piece(MyString, "linebreak") 'InterSystems
Println Piece(MyString, "linebreak", 2) 'One Memorial Drive
Println Piece(MyString, "linebreak", 3) 'Cambridge MA
Println Piece(MyString, "linebreak", 4,4) '02142
```

See Also

- Split function

Replace

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

```
Replace(string,find,replacewith[,start[,count[,compare]]])
```

Arguments

string	String expression containing substring to replace.
find	Substring being searched for. If found, it is replaced by *replacewith*.
replacewith	Replacement substring.
start	*Optional* — Position within *string* 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. Must be used in conjunction with *start*.
compare	*Optional* — Numeric value indicating the kind of comparison to use when evaluating substrings. See Description section for values. If omitted, the default value is 0, which means perform a binary comparison.

Description

The *compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a case-sensitive (binary) comparison.
vbTextCompare	1	Perform a not case-sensitive textual comparison.

Note: The not case-sensitive comparison works as expected for all Caché-supported locales.

The **Replace** function returns the following values:

If	Replace Returns
string is zero-length	A zero-length string ("").
find is zero-length	A copy of *string*.
replacewith is zero-length	A copy of *string* with all occurrences of *find* removed.
start > Len(*string*)	A zero-length string ("").
count is 0	A copy of *string*.

The return value of the **Replace** function is a substring, with substitutions made, that begins at the position specified by *start* and concludes at the end of the expression string.

Examples

The following example starts at the beginning of the string and replaces all (by default), or the specified number of *find* substrings:

```
Println Replace("To ski or not to ski","ski","be")
Println Replace("To ski or not to ski","ski","be",1,2)
Println Replace("To ski or not to ski","ski","be",1,1)
```

The following example starts at the specified location in the string and replaces all (by default), or the specified number of *find* substrings:

```
Println Replace("To ski or not to ski","ski","be",4,2)
Println Replace("To ski or not to ski","ski","be",4,1)
```

Note that the returned value is not the original string with substitutions; it is a substring that starts at the point specified by the fourth (*start*) parameter.

The following example performs binary and textual comparisons:

```
' A binary comparison starting at the beginning
' of the string. Returns "XXYXXPXXY".
Println Replace("XXpXXPXXp", "p", "Y")

' A textual comparison starting at position 3.
' Returns "YXXYXXY".
Println Replace("XXpXXPXXp", "p", "Y", 3, -1, 1)
```

Right

Returns or replaces a specified number of characters from the right end of a string.

```
Right(string,length)
Right(string,length)=value
```

Arguments

string	String expression from which the rightmost characters are returned.
length	Numeric expression that evaluates to an integer indicating how many characters from the end of *string* to return or replace. Fractional numbers are truncated to an integer. If *length* is greater than or equal to the number of characters in *string*, the entire string is returned (or replaced). No padding is performed. For *length*=0, see below.
value	An expression that evaluates to a string. Specifies the value used to either replace or append. If *length* is 0, *value* is appended to the end of *string*. If *length* is greater than zero, *value* replaces the specified character(s) at the end of *string*.

Description

The **Right** function can be used in three ways:

- To return a substring from the end (right end) of *string*. This uses the `Right(string,length)` syntax.
- To replace a substring from the end (right end) of *string*. The replacement substring may be the same length, longer, or shorter than the original substring. This uses the `Right(string,length)=value` syntax, with *value*>0.
- To append a substring to the end (right end) of *string*. This uses the `Right(string,length)=value` syntax, with *value*=0.

`Right(string,length)` returns the rightmost character(s) of *string*. The substring is determined by counting *length* characters backwards from the end (right end) of the string. If *length* is 0 or a negative number, **Right** returns the empty string (""). If you specify a *length* greater than the length of *string*, the entire string is returned.

`Right(string,length)=value` replaces the rightmost character(s) of *string* with *value*. The *length* argument specifies how many characters of *string* to replace by counting *length* characters backwards from the end (right end) of the string. If *length* is a negative number, *string* is unchanged. This is true even when *string* is the empty string (""). If *length* is greater than the length of *string*, *string* is replaced by *value*. If *length*=0, the *value* is appended to the end (right end) of *string*.

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

The **Left** function returns the specified number of characters from the beginning (left end) of a string. The **Mid** function returns the specified number of characters from a specified starting point within a string.

Examples

The following example uses the **Right** function to return the last seven characters of mystr, the last 99 characters (in this case, all of the characters), and the last 0 characters:

```
Dim mystr
mystr = "InterSystems"
Println "length 7:",Right(mystr,7)   ' Returns "Systems"
Println "length 99:",Right(mystr,99) ' Returns "InterSystems"
Println "length 0:",Right(mystr,0)   ' Returns ""
```

The following example uses the **Right** function to replace the last three characters of mystr with a seven-character string:

```
Dim mystr
mystr = "Interest"
PrintLn mystr
Right(mystr,3)="Systems"
PrintLn mystr
```

The following example deletes (replaces with the null string) the last three characters of mystr:

```
Dim mystr
mystr = "Interest"
PrintLn mystr
Right(mystr,3)=""
PrintLn mystr
```

The following example replaces all of the characters of mystr, because *length* is greater than the length of mystr:

```
Dim mystr
mystr = "Oracle"
PrintLn mystr
Right(mystr,99)="InterSystems"
PrintLn mystr
```

The following example appends a string to mystr. To append a string, *length* must be equal to zero (0):

```
Dim mystr
mystr = "Inter"
PrintLn mystr
Right(mystr,0)="Systems"
PrintLn mystr
```

The following example shows that a *length* less than 0 has no effect on mystr:

```
Dim mystr
Dim empstr
mystr = "InterSystems"
empstr = ""
PrintLn mystr
Right(mystr,-1)="Bongo"
PrintLn "string out:",mystr
Right(empstr,-1)="BongoSystems"
PrintLn "string out:",empstr
```

See Also

- Left function
- Len function
- Mid function

Rnd

Returns a random number.

```
Rnd[(number)]
```

Arguments

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

Description

The **Rnd** function returns a value less than 1 but greater than or equal to 0. The number of digits in this number is platform-dependent. Trailing zeros are deleted.

Rnd generates a pseudo-random number by calculating successive numbers from a seed number supplied by the *number* argument. Thus, the value of *number* determines how **Rnd** generates a random number. **Rnd** with no argument or **Rnd** with a positive *number* generate random numbers from a randomized seed. Therefore, successive executions of **Rnd** with the same positive *number* return different values. However, if *number* is zero or a negative number, each successive call to the **Rnd** function uses the same seed, and thus generates a predictable value.

If Number Is	Rnd Generates
Less than zero	The same number every time, using *number* as the seed. Thus, for example, -7 always generates .5976062.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated random number.
Not supplied	The next random number in the sequence.

To maximize randomness, use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer. Then call **Rnd**.

Notes

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.

Examples

The following example generates twenty random numbers.

```
   For I = 1 To 20
   Println Rnd
   Next
Println "Done"
```

The following example generates a random integer in the range 1 through 10, inclusive:

```
Dim upperbound,lowerbound
upperbound = 10
lowerbound = 1
Println Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```

The following example shows the effects of specifying 0 as the *number* argument:

```
   For I = 1 To 10
   Println Rnd
   Println Rnd(0)
   Next
Println "Done"
```

In this case, the argumentless **Rnd** generates a random number, and the **Rnd(0)** repeats the most-recently-generated random number.

The following example shows the effects of specifying a negative number as the *number* argument:

```
   For I = 1 To 10
   Println Rnd
   Println Rnd(-7)
   Next
Println "Done"
```

In this case, the first argumentless **Rnd** generates a random number, and the **Rnd(-7)** calculates its corresponding value and provides this as the seed for the next random number. Thus in the above example, the first call to **Rnd** is actually random; all subsequent calls are based on the seed of -7, and therefore repeat predictably in each loop.

See Also

- Randomize statement

Round

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

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

Arguments

expression	The numeric expression being rounded.
numdecimalplaces	*Optional* — Number indicating how many places to the right of the decimal are included in the rounding. If omitted, *expression* rounded to an integer is returned.

Description

The **Round** function always rounds the number 5 up. Leading and trailing zeros are deleted.

Round returns *expression* rounded to:

- An integer if the *numdecimalplaces* argument is omitted, or specified as 0, the empty string (""), or a negative number.
- The specified number of decimal places (excluding trailing zeros). A fractional value for *numdecimalplaces* is truncated to an integer.
- The actual number of decimal places (excluding trailing zeros) if the *numdecimalplaces* argument is greater than or equal to the actual number of decimal places

Examples

The following example uses the **Round** function to round a number to three decimal places:

```
Dim MyVar, pi
pi = 3.14159
MyVar = Round(pi,3) ' MyVar contains 3.142.
Println MyVar
```

See Also

- Int, Fix functions

Second

Returns a whole number between 0 and 59, inclusive, representing the second of the minute.

```
Second(time)
```

Arguments

The time argument is any expression that can represent a time. This includes a time value such as “12:30” or “1:27:55”, a time/date value such as “11/12/1999 12:33:00”, or a date value such as “11/12/1999”. If only a date is specified, the time defaults to 00:00:00. Fractional seconds are permitted, but ignored; they are truncated, not rounded.

Examples

The following example uses the **Second** function to obtain the second from the current time:

```
Dim MyTime, MySec
MyTime = Now
MySec = Second(MyTime)
Println MySec
```

The following example returns a second value of 59:

```
Dim MySec
MySec = Second("13:59:59.999")
Println MySec
```

See Also

- Day function
- Hour function
- Minute function
- Now function
- Time function

Sgn

Returns an integer indicating the sign of a number.

```
Sgn(number)
```

Arguments

The number argument can be any valid numeric expression.

Description

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

If Number Is	Sgn Returns
Greater than zero	1
Equal to zero	0
Less than zero	–1

The sign of the *number* argument determines the return value of the **Sgn** function. If *number* is the empty string ("") or a non-numeric value, **Sgn** returns 0. **Sgn** resolves multiple sign values; for example, `--7` is equivalent to `+7`, and thus returns 1.

Examples

The following example uses the **Sgn** function to determine the sign of a number:

```
Dim MyVar1, MyVar2, MyVar3
MyVar1 = 12: MyVar2 = -2.4: MyVar3 = 0
Println Sgn(MyVar1)    ' Returns 1.
Println Sgn(MyVar2)    ' Returns -1.
Println Sgn(MyVar3)    ' Returns 0.
Println Sgn("")        ' Returns 0.
Println Sgn("a")       ' Returns 0.
```

See Also

- Abs function

Sin

Returns the sine of an angle.

```
Sin(number)
```

Arguments

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Description

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.

Examples

The following example uses the **Sin** function to return the sine of an angle:

```
Dim MyAngle
MyAngle = 1.3          ' Define angle in radians.
Println Sin(MyAngle)
```

The following example uses the **Sin** function to return the cosecant of an angle:

```
Dim MyAngle, MyCosecant
MyAngle = 1.3                   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
Println MyCosecant
```

See Also

- Atn function
- Cos function
- Tan function
- Derived Math Functions

Space

Returns a string consisting of the specified number of spaces.

```
Space(number)
```

Arguments

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

Examples

The following example uses the **Space** function to return a string consisting of a specified number of spaces:

```
Dim MyString
MyString = Space(10)   ' Defines a string of 10 spaces.
Println "Hello" & MyString & "World"
                       ' Insert 10 spaces between two strings.
```

See Also

- String function

Split

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

```
Split(string[,delimiter[,count[,compare]]])
```

Arguments

string	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. Usually a single character, but can be a multi-character string. 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.
count	*Optional* — Number of substrings to be returned. If -1, or an integer equal to or greater than number of substrings in *string*, all substrings are returned.
compare	*Optional* — Numeric value indicating the kind of comparison to use when evaluating substrings. See Description section for values.

Description

The **Split** function splits a string into array elements. The **Join** function does the opposite; it joins array elements into a string.

The *compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Split and For Each

A **Split** function cannot be directly used as an argument of a **For Each...Next** statement. You must first assign the **Split** return value to an array variable. You can then specify this array variable as the *group* argument of the **For Each...Next** statement.

Examples

The following example uses the **Split** function to return an array from a string. By default, it uses the blank space as the string delimiter character.

```
Dim MyString,MyArray
MyString = "Caché is fun!"
MyArray = Split(MyString)
Println MyArray(0)   ' contains "Caché".
Println MyArray(1)   ' contains "is".
Println MyArray(2)   ' contains "fun!".
```

The following example demonstrates the *delimiter* argument. It uses a two-character delimiter. The string is split at each occurrence of the “aa” delimiter. A single “a” is treated as a literal, as is the third “a” in the substring “aaa”.

```
Dim MyString,MyArray
MyString = "Cachéaaisaaafun!"
MyArray = Split(MyString,"aa")
Println MyArray(0)   ' contains "Caché".
Println MyArray(1)   ' contains "is".
Println MyArray(2)   ' contains "afun!".
```

The following example demonstrates the use of the *count* argument. It returns only the specified number of substrings (in this case, 2) into array elements. Note that in this case only part of the string is returned.

```
Dim MyString, MyArray, Msg
MyString = "Caché;is;fun!"
MyArray = Split(MyString,";",2)
Println MyArray(0)   ' contains "Caché".
Println MyArray(1)   ' contains "is".
Println MyArray(2)   ' contains "".
```

The following example demonstrates the *compare* argument. It shows the difference between a binary comparison and a textual comparison. In a binary comparison, only the lowercase “x” is considered to be the delimiter; in a textual comparison, both “x” and “X” are treated as the delimiter character.

```
Dim MyString,MyArray
MyString = "CachéXisxfun!"
MyArray = Split(MyString,"x",-1,0)
Println "Binary: ",MyArray(0)   ' contains "CachéXis".
Println "Binary: ",MyArray(1)   ' contains "fun!".
MyArray = Split(MyString,"x",-1,1)
Println "Textual: ",MyArray(0)   ' contains "Caché".
Println "Textual: ",MyArray(1)   ' contains "is".
Println "Textual: ",MyArray(2)   ' contains "fun!".
```

See Also

- Join function

Sqr

Returns the square root of a number.

```
Sqr(number)
```

Arguments

The *number* argument can be any valid numeric expression greater than or equal to 0. You cannot return the square root of a negative number. Attempted to do so results in a runtime error.

Examples

The following example uses the **Sqr** function to calculate the square roots of the integers 0 through 16:

```
For x = 0 To 16
Println "Square root of ",x," = ",Sqr(x)
Next
```

The following example uses the **Sqr** function to calculate the square root of pi:

```
pi = 4 * Atn(1)
Println "Square root of pi = ",Sqr(pi)
```

See Also

- Derived Math Functions

StrComp

Returns a value indicating the result of a string comparison.

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

Arguments

string1	Any valid string expression.
string2	Any valid string expression.
compare	*Optional* — Numeric value indicating the kind of comparison to use when evaluating strings. If omitted, a binary comparison is performed. See Description section for values.

Description

The **StrComp** function compares two strings character-by-character and returns a value when the first non-matching character is encountered, or when the end of the string has been encountered. **StrComp** returns one of the following values:

- 0 if the two strings are identical, or if *compare*=1 and the strings differ only in the case of letters.
- 1 if *string1* contains a non-matching character that has a higher ANSI character code value than the corresponding character in *string2*. If *compare*=1 letters that differ in case are treated as identical. If *string1* is longer than *string2*, **StrComp** returns 1.
- -1 if *string1* contains a non-matching character that has a lower ANSI character code value than the corresponding character in *string2*. If *compare*=1 letters that differ in case are treated as identical. If *string1* is shorter than *string2*, **StrComp** returns -1.

The *compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison. This is the default.
vbTextCompare	1	Perform a textual comparison. Uppercase and lowercase letters are equivalent.

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

Examples

The following example compares two strings that differ only in the case of letters:

```
Println "default:  ",StrComp("abcd","ABCD")    ' Returns 1
Println "binary:   ",StrComp("abcd","ABCD",0)  ' Returns 1
Println "textual:  ",StrComp("abcd","ABCD",1)  ' Returns 0
Println "default:  ",StrComp("ABCD","abcd")    ' Returns -1
Println "binary:   ",StrComp("ABCD","abcd",0)  ' Returns -1
Println "textual:  ",StrComp("ABCD","abcd",1)  ' Returns 0
```

The following example compares two strings that differ only in length:

```
Println "binary:  ",StrComp("abcde","abcd",0)  ' Returns 1
Println "textual: ",StrComp("abcde","abcd",1)  ' Returns 1
Println "binary:  ",StrComp("abcd","abcde",0)  ' Returns -1
Println "textual: ",StrComp("abcd","abcde",1)  ' Returns -1
```

String

Returns a repeating character string of the length specified.

```
String(length,character)
```

Arguments

length	A positive integer specifying the length of the generated string. A zero length returns an empty string; a negative length returns no value. Fractional numbers are truncated.
character	Character code specifying the character, or a string expression whose first character is used to build the return string.

Description

The **String** function returns a string consisting of a single repeated character; *length* specifies the number of times to repeat the character. Use the **Space** function to return a string consisting of blank spaces.

A character code value must evaluate to a positive integer in the range 0 to 255 (inclusive). If you specify a character code for *character* greater than 255, **String** converts the number to a character code in the range 0 through 255 using the formula:

```
character Mod 256
```

For 16-bit characters, you can use the **Chr** function, as shown in the example below.

Examples

The following example uses the **String** function to return repeating character strings of the length specified:

```
Println String(5,"*")       ' Returns "*****".
Println String(5,42)        ' Returns "*****".
Println String(10,"ABC")    ' Returns "AAAAAAAAAA".
```

The following example uses character code values and the **Chr** function to specify the repeating character. Note that you must use the **Chr** function for character codes beyond 255.

```
Println String(10,65)       ' Returns "AAAAAAAAAA"
Println String(10,321)      ' Returns "AAAAAAAAAA"
Println String(10,577)      ' Returns "AAAAAAAAAA"
Println String(5,Chr(960))  ' Returns five pi symbols
```

See Also

- Chr function
- Space function

StrReverse

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

```
StrReverse(string1)
```

Arguments

The *string1* argument is the string whose characters are to be reversed. If *string1* is a zero-length string (""), a zero-length string is returned.

Examples

The following example uses the **StrReverse** function to return a string in reverse order:

```
Dim RevStr
RevStr = StrReverse("Caché") ' RevStr contains "éhcaC"
Println "backwards: ",RevStr
Println "forewards: ",StrReverse(RevStr)
```

Tan

Returns the tangent of an angle.

```
Tan(number)
```

Arguments

The *number* argument can be any valid numeric expression that expresses an angle in radians.

Description

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.

Examples

The following example uses the **Tan** function to return the tangent of an angle:

```
Dim MyAngle
MyAngle = 1.3           ' Define angle in radians.
Println Tan(MyAngle)    ' Calculate tangent.
```

The following example uses the **Tan** function to return the cotangent of an angle:

```
Dim MyAngle, MyCotangent
MyAngle = 1.3                   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)  ' Calculate cotangent.
Println MyCotangent
```

See Also

- Atn function
- Cos function
- Sin function
- Derived Math Functions

Time

Returns the current system time.

```
Time
```

Arguments

none

Description

The **Time** function returns the current time in a format such as the following:

```
00:00:00PM
```

In this case, time is represented by a 12–hour clock with an AM/PM indicator and no fractional seconds. The exact display format depends on your system configuration. Leading zeros are displayed.

Examples

The following example uses the **Time** function to return the current system time:

```
Dim MyTime
MyTime = Time
Println MyTime
```

See Also

- Basic: Date function, Now function
- ObjectScript: $HOROLOG special variable
- SQL: NOW function

TimeConvert

Converts time between internal and external formats.

```
TimeConvert(time,vbToInternal)
TimeConvert(time,vbToExternal)
```

Arguments

time	The time to be converted. An external time is represented as a string, such as “10:23:54”. An internal time is represented as a numeric value, which is the second part of the Caché $H date/time format.
vbToInternal	This keyword specifies converting an external time to internal ($H) format.
vbToExternal	This keyword specifies converting an internal time ($H format) to external time format.

Description

The **TimeConvert** function returns an external time in the following format:

```
00:00:00
```

Leading zeros are displayed.

The **TimeConvert** function returns an internal date/time in the following format:

```
sssss.ff
```

Where “sssss” is the time count (number of elapsed seconds in the specified day), and “ff” is optional fractional seconds. Fractional seconds are preserved in converting from external to internal format; fractional seconds are truncated when converting from internal to external format. For further details, see $HOROLOG in the *Caché ObjectScript Reference*.

Examples

The following example takes an external time value, converts the time to an internal format ($HOROLOG) value, then converts this internal value back to an external format time.

```
Dim GetDT,InTime, ExTime
GetDT = "21:45:23.99"
Println GetDT
InTime = TimeConvert(GetDT,vbToInternal)
Println InTime
ExTime = TimeConvert(InTime,vbToExternal)
Println ExTime
```

See Also

- DateConvert function
- DateTimeConvert function

Timer

Returns the number of seconds that have elapsed since midnight UTC.

```
Timer
```

Arguments

none

Description

Timer is commonly used to determine elapsed time. However, because **Timer** resets to zero at midnight UTC, a robust timer program cannot simply subtract the start time from the end time.

Timer returns the elapsed number of seconds since midnight in Coordinated Universal Time (UTC), which is independent of time zone. Consequently, **Timer** provides a time value that is uniform across time zones. This may differ from the local time value. The **Timer** returned value is a decimal numeric value that counts the time in seconds and fractions thereof. The number of digits in the fractional seconds may vary from zero to nine, depending on the precision of your computer's time-of-day clock. On Windows systems the fractional precision is three decimal digits; on UNIX® systems it is six decimal digits. **Timer** suppresses leading and trailing zeroes. If the fractional portion is exactly zero, the trailing decimal point is also suppressed.

Example

The following example compares two **Timer** function values to determine the time it takes to iterate a **For...Next** loop 50 times. In this example, each iteration through the loop prints out the ASCII character set. (For the purpose of demonstration, a **Sleep** statement is included so that the elapsed time is not smaller than the available fractional precision.) The before and after timer values are compared, and the elapsed time displayed. The Else clause is provided to handle the midnight reset situation.

```
Dim StartTime,EndTime,TimeIt
StartTime = Timer
For I = 1 To 50
   Sleep .05
   x = 32
   Print I
   For N = 1 To 94
   Print Chr(x)
   x = x + 1
   Next
   Println ""
Next
EndTime = Timer
Println "start time: ",StartTime
Println "end time: ",EndTime
If EndTime >= StartTime Then
  TimeIt = EndTime - StartTime
Else
  EndTime = EndTime + 86400
  TimeIt = EndTime - StartTime
End If
 Println "elapse time: ",TimeIt
```

See Also

- Randomize statement
- Sleep statement

TimeSerial

Returns the time for a specific hour, minute, and second.

```
TimeSerial(hour,minute,second)
```

Arguments

hour	Number between 0 (12:00AM) and 23 (11:00PM), inclusive, or a numeric expression the evaluates to a number in the range 0 through 23.
minute	Any numeric expression.
second	Any numeric expression.

Description

To specify a time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should be in the accepted 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. When any argument exceeds the accepted 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. However, you cannot specify an hour value greater than 23.

The **TimeSerial** function, by default, returns a 12-hour clock time value with an AM or PM suffix. Leading zeros are displayed. Fractional seconds are truncated.

Examples

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 5:45:00AM

```
Dim MyTime
MyTime = TimeSerial(12 - 6, -15, 0) ' Returns 5:45:00 AM.
Println MyTime
```

See Also

- DateSerial function
- Hour function
- Minute function
- Now function
- Second function

Traverse

Traverses an array and returns the next subscript.

```
Traverse(varname[,direction[,target]])
```

Arguments

varname	The name of a variable representing an array. *varname* must specify a subscript level.
direction	*Optional* — Numeric value, which specifies either forward (1) or backward (-1). If this parameter is omitted the **Traverse** function will move forward to the next subscript.
target	*Optional* — Variable which contains the data of the resulting node.

Description

The **Traverse** function returns the name of the next or previous subscript on the specified subscript level. Using the optional, *target* argument you can also return the data value of the located subscript.

To start a search from the beginning of the current level, specify a empty string ("") for the subscript. The following example returns the first subscript on the first subscript level:

```
subscript = Traverse(^Person(""))
```

When the **Traverse** reaches the end of the subscripts for the given level, it returns an empty string ("") .

The following example demonstrates how to use the **Traverse** function within a loop:

```
subscript = ""
subscript = Traverse(^Person(subscript))
While subscript <> ""
  subscript = Traverse(^Person(subscript))
  Println subscript
wend
```

If a *target* variable is specified and the node is defined (vbHasValue) the *target* variable will contain the data for this node. If the resulting node does not have a value the value of the *target* variable is unchanged.

Caché Basic provides two constants, vbForward and vbBackward to specify the direction.

Examples

The following example demonstrates the use of the **Traverse** function to return the subscript name of the next node. The first **Traverse** specifies the empty string as the subscript, and returns the name of the first subscript (“A”) in the array. The second and third **Traverse** functions specify a subscript name and return the name of the next subscript. The fourth **Traverse** specifies a subscript of “C” and a *direction* of -1 (backwards); it returns the name of the previous subscript (“B”). The final **Traverse** specifies the empty string as the subscript, and a *direction* of -1; it returns the name of the final subscript (the first subscript reading backwards from the end of the array), in this case “D”.

```
array("A") = "A node"
array("B") = "B node"
array("B", 1) = "B,1 node"
array("B", 2) = "B,2 node"
array("C", 1) = "C node"
array("D") = "D node"

Println Traverse(array(""))  ' prints A
Println Traverse(array("A")) ' prints B
Println Traverse(array("B")) ' prints C
Println Traverse(array("C"),-1) ' prints B
Println Traverse(array(""),-1) ' prints D
```

The following example demonstrates the use of the *target* argument. The first myString contains the data value of the “A” node. The second myString references a node (“C”) which contains on data value at this level. In this case, myString continues to contain its previous value.

```
array("A") = "A node"
array("B") = "B node"
array("C", 1) = "C1 node"
array("D") = "D node"

Println Traverse(array(""),1,myString)  ' prints A
Println myString                        ' prints A node
Println Traverse(array("B"),1,myString) ' prints C
Println myString                        ' prints A node
```

See Also

- Exists function

LTrim, RTrim, and Trim

Returns a copy of a string without leading spaces (LTrim), trailing spaces (RTrim), or both leading and trailing spaces (Trim).

```
LTrim(string)
RTrim(string)
Trim(string)
```

Arguments

The *string* argument is any valid string expression.

Examples

The following example uses the **LTrim**, **RTrim**, and **Trim** functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively:

```
Dim MyVar
MyVar = LTrim("  Caché ")  'MyVar contains "Caché ".
Println Len(MyVar),":",MyVar,":"
MyVar = RTrim("  Caché ")  'MyVar contains "  Caché".
Println Len(MyVar),":",MyVar,":"
MyVar = Trim("  Caché ")   'MyVar contains "Caché".
Println Len(MyVar),":",MyVar,":"
```

See Also

- Left function
- Right function

UCase

Returns a string that has been converted to uppercase.

```
UCase(string)
```

Arguments

The *string* argument is any valid string expression.

Description

Only lowercase letters are converted to uppercase. Uppercase letters and non-letter characters remain unchanged.

Examples

The following example uses the **UCase** function to convert lowercase letters to uppercase:

```
Dim MyString
MyString = "Caché from InterSystems"
Println UCase(MyString) ' Returns "CACHÉ FROM INTERSYSTEMS"
```

The following example converts the first four letters of the Greek alphabet from lowercase to uppercase:

```
Dim MyString
MyString = Chr(945)&Chr(946)&Chr(947)&Chr(948)
Println MyString
Println UCase(MyString)
```

(Note that the above example requires a Unicode installation of Caché.)

See Also

- LCase function

Unlock

Releases a logical lock on a variable name.

```
Unlock(varname)
```

Arguments

varname	Name of the variable to be unlocked.

Description

Each time a lock is obtained on a *varname* a lock count is incremented. **Unlock** decrements this count. Only when the lock count falls to zero will the logical lock be released. For this reason, you should balance each call to **Lock** with a corresponding call to **Unlock**.

Examples

The following example uses the **Lock** function to obtain a logical lock on a global variable name (glvn) with a timeout of 10 seconds, and then uses the **Unlock** function to release the logical lock.

```
If Lock(^PatientData(PatientID),10) = True Then
  Println "Got the Lock"
  Unlock(^PatientData(PatientID))     'Release the logical lock
Else
  Println "Couldn't get the lock"
End If
```

See Also

- Lock function

Weekday

Returns a whole number representing the day of the week.

```
Weekday(weekday[,firstdayofweek])
```

Arguments

weekday	Any expression that can represent a date.
firstdayofweek	*Optional* — A constant that specifies the first day of the week. If omitted, **vbSunday** is assumed. See Description section for values.

Description

The **Weekday** function returns an integer between 1 and 7 (inclusive) specifying the day of the week represented by *weekday*. The first day of the week is, by default, Sunday, or the current NLS day of week setting overriding this default system-wide.

The *firstdayofweek* argument can be used to set the first day of the week for this statement to the day of your choosing. The *firstdayofweek* argument can have the following values:

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

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

Examples

The following example uses the **Weekday** function to obtain the default day of the week for the specified date:

```
MyDay0 = Weekday("11/1/2005")
Println MyDay0
```

Nov. 1, 2005 is a Tuesday, so `Weekday("11/1/2005")` returns 3.

The following example shows the effects of the *firstdayofweek* argument:

```
MyDay1 = Weekday("11/1/2005",vbSunday)
MyDay2 = Weekday("11/1/2005",vbMonday)
MyDay3 = Weekday("11/1/2005",vbTuesday)
MyDay4 = Weekday("11/1/2005",vbWednesday)
MyDay5 = Weekday("11/1/2005",vbThursday)
MyDay6 = Weekday("11/1/2005",vbFriday)
MyDay7 = Weekday("11/1/2005",vbSaturday)
Println "Day is: ",MyDay1," Week begins Sunday"
Println "Day is: ",MyDay2," Week begins Monday"
Println "Day is: ",MyDay3," Week begins Tuesday"
Println "Day is: ",MyDay4," Week begins Wednesday"
Println "Day is: ",MyDay5," Week begins Thursday"
Println "Day is: ",MyDay6," Week begins Friday"
Println "Day is: ",MyDay7," Week begins Saturday"
```

See Also

- Date function
- Day function
- Month function
- Now function
- Year function

WeekdayName

Returns a string indicating the specified day of the week.

```
WeekdayName(weekday[,abbreviate[,firstdayofweek]])
```

Arguments

weekday	An integer 1 through 7, inclusive, designating the day of the week. The numeric value of each day depends on the *firstdayofweek* setting.
abbreviate	*Optional* — Boolean value that indicates if the weekday name is to be returned fully spelled out, or abbreviated. If omitted, the default is **False**, which means that the weekday name is returned fully spelled out.
firstdayofweek	*Optional* — Numeric value indicating the first day of the week. See Description section for values.

Description

The **WeekdayName** function returns a day name corresponding to the day of the week represented by *weekday*. The first day of the week is, by default, Sunday, or the current NLS day of week setting overriding this default system-wide.

The *firstdayofweek* argument can be used to set the first day of the week for this statement to the day of your choosing. The *firstdayofweek* argument can have the following values:

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

Examples

The following example uses the **WeekDay** and **WeekDayName** functions to return the name of the day of the week for the specified date:

```
Dim WDayNum,WDayName
WDayNum = Weekday("11/1/2005")
Println WDayNum
WDayName = WeekdayName(3)
Println WDayName
```

November 1, 2005 is a Tuesday.

See Also

- MonthName function

- WeekDay function

Year

Returns the year as a four-digit integer.

```
Year(date)
```

Arguments

The *date* argument is any expression that represents a date as a string.

Description

The **Year** function locates and returns the year portion of a date string as a four-digit integer. The **Year** function accepts blanks, slashes (/), hyphens (-), or commas (,) (in any combination) as date component separators.

The **Year** function locates the year portion, by position, as the third portion of a date. For example, “9/27/2005” or “September 27, 2005” or “27 September 2005”. It does not validate the day or month components of the date string.

If the **Year** function is unable to identify a year portion of the string, it returns a default value of “2000”. If the year portion is provided as a two-digit year, the **Year** function returns a four-digit year. If the two-digit year is 29 or less, it supplies ”20” for the missing century digits. If the two-digit year is greater than 29, it supplies 19” for the missing century digits.

Examples

The following example uses the **Year** function to return the current year:

```
Dim CurrYear
CurrYear = Year(Date)
Print CurrYear
```

The following example uses the **Year** function to obtain the year from a series of specified dates. In every case except the last, it returns the string “2005”. In the last case, the third portion of the string cannot be parsed as a year; this **Year** function instead returns the default value “2000”.

```
Dim YearA, YearB, YearC, YearD, YearE,YearF,YearG
YearA = Year("August 12 2005")
YearB = Year("Agosto 12 2005 11:35am")
YearC = Year("Aug 12 05 11:35am")
YearD = Year("12 Agosto 2005")
YearE = Year("8/12/2005")
YearF = Year("8-12-05 11:35am")
YearG = Year("August 12 11:35am")
Println YearA
Println YearB
Println YearC
Println YearD
Println YearE
Println YearF
Println YearG
```

See Also

- Basic: Date function, Day function, Hour function, Minute function, Month function, Now function, Second function, Weekday function,
- ObjectScript: $ZDATE function
- SQL: YEAR function

Caché Basic Operators

Operator Summary

A list of Caché Basic operators by type.

Arithmetic Operators

- Addition: + Operator
- Subtraction: – Operator
- Multiplication: * Operator
- Division: / Operator
- Integer Division: \ Operator
- Exponentiation: ^ Operator
- Modulus: Mod Operator

Assignment Operator

- = Operator

Comparison Operators

- Less Than/Greater Than: Comparison Operators
- Object Reference Comparison: Is Operator

Concatenation Operator

- & Operator

Logical Operators

- And Operator
- Not Operator
- Or Operator
- Xor Operator
- Eqv Operator
- Imp Operator

Bitwise Logical Operators

- BitAnd Operator
- BitNot Operator
- BitOr Operator
- BitXor Operator
- BitEqv Operator

- BitImp Operator

Operator Precedence

Operator Precedence

Description

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. 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	Bitwise
Negation (-)	Equality (=)	Not	BitNot
Exponentiation (^)	Inequality (<>)	And	BitAnd
Multiplication and division (*, /)	Less than (<)	Or	BitOr
Integer division (\)	Greater than (>)	Xor	BitXor
Modulus arithmetic (Mod)	Less than or equal to (<=)	Eqv	BitEqv
Addition and subtraction (+, -)	Greater than or equal to (>=)	Imp	BitImp
String concatenation (&)	Is	&	

Note: Caché Basic gives the And logical operator precedence over the Or logical operator. This order of evaluation differs from ObjectScript and Caché MultiValue Basic, both of which give And and Or equal precedence.

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation operator (&) is not an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. 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.

See Also

- Is Operator
- Operator Summary

Addition Operator (+)

Used to sum two numbers.

```
result = expression1 + expression2
```

Arguments

The + operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

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

When you use the + operator, you may not be able to determine whether addition or string concatenation will occur.

The underlying subtype of the expressions determines the behavior of the + operator in the following way:

If	Then
Both expressions are numeric	Add.
Both expressions are strings	Concatenate.
One expression is numeric and the other is a string	Add.

Notes

If both expressions are Empty, *result* is an Integer subtype. However, if only one expression is Empty, the other expression is returned unchanged as *result*.

See Also

- Operator Summary
- Operator Precedence
- Subtraction Operator (–)

Subtraction 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
```

Arguments

The – operator syntax has these parts:

result	Any numeric variable.
number	Any numeric expression.
number1	Any numeric expression.
number2	Any numeric expression.

Description

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.

If an expression is Empty, it is treated as if it were 0.

See Also

- Operator Summary
- Operator Precedence
- Addition Operator (+)

Mod Operator

Used to divide two numbers and return only the remainder.

```
result = number1 Mod number2
```

Arguments

The **Mod** operator syntax has these parts:

result	Any numeric variable.
number1	Any numeric expression.
number2	Any numeric expression.

Description

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 (which is result) equals 5.

```
A = 19 Mod 6.7
```

Any expression that is Empty is treated as 0.

See Also

- Operator Precedence
- Operator Summary

Multiplication Operator (*)

Used to multiply two numbers.

```
result = number1*number2
```

Arguments

The * operator syntax has these parts:

result	Any numeric variable.
number1	Any numeric expression.
number2	Any numeric expression.

Description

If an expression is Empty, it is treated as if it were 0.

See Also

- Operator Summary
- Operator Precedence
- Division Operator (/)

Division Operator (/)

Used to divide two numbers and return a floating-point result.

```
result = number1/number2
```

Arguments

The / operator syntax has these parts:

result	Any numeric variable.
number1	Any numeric expression.
number2	Any numeric expression.

Description

Any expression that is Empty is treated as 0.

See Also

- Operator Summary
- Operator Precedence
- Multiplication Operator (*)
- Integer Division Operator (\)

Integer Division Operator (\)

Used to divide two numbers and return an integer result.

```
result = number1 \ number2
```

Arguments

The \ operator syntax has these parts:

result	Any numeric variable.
number1	Any numeric expression.
number2	Any numeric expression.

Description

Before division is performed, numeric expressions are rounded to **Byte**, **Integer**, or **Long** subtype expressions.

Any expression that is Empty is treated as 0.

See Also

- Operator Summary
- Operator Precedence
- Multiplication Operator (*)
- Division Operator (/)

Exponent Operator (^)

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

```
result = number ^ exponent
```

Arguments

result	Any numeric variable.
number	Any numeric expression.
exponent	Any numeric expression. An exponent value is required.

Description

The caret (^) is used as the exponentiation operator.

Note: The exponent operator should not be confused with the base-10 exponent symbol. An uppercase letter “E”, or lowercase letter “e” can be used as a base-10 exponent (scientific notation) symbol in a numeric literal. These letters cannot be used as operators.

The *number* argument 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.

Caché Basic exponentiation is functionally identical to ObjectScript exponentiation. For details on valid argument values and the value returned for specific combinations of argument values, see Exponentiation Operator in the “Operators and Expressions” chapter of *Using Caché ObjectScript*.

Example

The following example shows valid uses of the exponent operator (^) and the base-10 exponent symbol (E). Note that the usage `x E y` is not valid, because E is a numeric literal character, not an operator.

```
SET x=3
SET y=4
Println  x ^ y   ' Returns 81
Println  3E4     ' Returns 30000
```

See Also

- Operator Summary
- Operator Precedence
- Multiplication Operator (*)

Assignment Operator (=)

Used to assign a value to a variable or property.

```
variable = value
```

Arguments

The = operator syntax has these parts:

variable	Any variable or any writable property.
value	Any numeric or string literal, constant, or expression.

Description

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 runtime.

See Also

- Operator Summary
- Operator Precedence
- Set Statement

Comparison Operators

Used to compare expressions.

```
result = expression1 comparisonoperator expression2
result = object1 Is object2
```

Arguments

Comparison operators have these parts:

result	Any numeric variable.
expression	Any expression.
comparisonoperator	Any comparison operator.
object	Any object name.

Description

The Is operator has specific comparison functionality that differs from the operators in the following table. The following table contains a list of the comparison operators and the conditions that determine whether result is True or False:

Operator	Description	True If	False if
<	Less than	expression1 < expression2	expression1 >= expression2
<=	Less than or equal to	expression1 <= expression2	expression1 > expression2
>	Greater than	expression1 > expression2	expression1 <= expression2
>=	Greater than or equal to	expression1 >= expression2	expression1 < expression2
=	Equal to	expression1 = expression2	expression1 <> expression2
<>	Not equal to	expression1 <> expression2	expression1 = expression2

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 expressions are compared or what results from the comparison, depending on the underlying subtype:

If	Then
Both expressions are numeric	Perform a numeric comparison.
Both expressions are strings	Perform a string comparison.
One expression is numeric and the other is a string	The numeric expression is less than the string expression.

Concatenation Operator (&)

Used to force string concatenation of two expressions.

```
result = expression1 & expression2
```

Arguments

The & operator syntax has these parts:

result	Any variable.
expression1	Any expression.
expression2	Any expression.

Description

Whenever an expression is not a string, it is converted to a **String** subtype. Any expression that is Empty is also treated as a zero-length string.

See Also

- Operator Summary
- Operator Precedence

Is Operator

Used to compare two object reference variables.

```
result = object1 Is object2
```

Arguments

The **Is** operator syntax has these parts:

result	Any numeric variable.
object1	Any object name.
object2	Any object name.

Description

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
```

See Also

- Operator Summary
- Operator Precedence
- And operator
- Not operator
- Xor operator

And Operator

Used to perform a logical conjunction on two expressions.

```
result = expression1 And expression2
```

Arguments

The **And** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

If, and only 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	If *expression2* Is	The *result* Is
True	True	True
True	False	False
False	True	False
False	False	False

See Also

- Operator Summary
- Operator Precedence
- BitAnd operator
- Not operator
- Or operator
- Xor operator

BitAnd Operator

Used to perform a bitwise conjunction on two numeric expressions.

```
result = expression1 BitAnd expression2
```

Arguments

The **BitAnd** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The **BitAnd** 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	If bit in *expression2* is	The *result* is
0	0	0
0	1	0
1	0	0
1	1	1

See Also

- Operator Summary
- Operator Precedence
- And operator
- BitNot operator
- BitOr operator
- BitXor operator

Eqv Operator

Used to perform a logical equivalence on two expressions.

```
result = expression1 Eqv expression2
```

Arguments

The **Eqv** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The *result* is determined according to the following table:

If *expression1* is	If *expression2* is	The *result* is
True	True	True
True	False	False
False	True	False
False	False	True

See Also

- Operator Summary
- Operator Precedence
- BitEqv operator
- And operator
- Not operator
- Xor operator

BitEqv Operator

Used to perform a bitwise equivalence on two numeric expressions.

```
result = expression1 BitEqv expression2
```

Arguments

The **BitEqv** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The **BitEqv** 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	If bit in *expression2* is	The *result* is
0	0	1
0	1	0
1	0	0
1	1	1

See Also

- Operator Summary
- Operator Precedence
- Eqv operator
- BitAnd operator
- BitNot operator
- BitXor operator

Imp Operator

Used to perform a logical implication on two expressions.

```
result = expression1 Imp expression2
```

Arguments

The **Imp** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The following table illustrates how *result* is determined:

If *expression1* is	If *expression2* is	The *result* is
True	True	True
True	False	False
False	True	True
False	False	True

See Also

- Operator Summary
- Operator Precedence
- BitImp operator
- And operator
- Not operator
- Xor operator

BitImp Operator

Used to perform a bitwise implication on two numeric expressions.

```
result = expression1 BitImp expression2
```

Arguments

The **BitImp** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The **BitImp** 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	If bit in *expression2* is	The *result* is
0	0	1
0	1	1
1	0	0
1	1	1

See Also

- Operator Summary
- Operator Precedence
- Imp operator
- BitAnd operator
- BitNot operator
- BitXor operator

Not Operator

Used to perform logical negation on an expression.

```
result = Not expression
```

Arguments

The **Not** operator syntax has these parts:

result	Any numeric variable.
expression	Any expression.

Description

The following table illustrates how *result* is determined:

If *expression* is	Then *result* is
True	False
False	True

See Also

- Operator Summary
- Operator Precedence
- BitNot operator
- And operator
- Or operator
- Xor operator

BitNot Operator

Used to perform bitwise negation on a numeric expression.

```
result = BitNot expression
```

Arguments

The **BitNot** operator syntax has these parts:

result	Any numeric variable.
expression	Any expression.

Description

The **BitNot** operator inverts the bit values of any variable and sets the corresponding bit in *result* according to the following table:

Bit in *expression*	Bit in *result*
0	1
1	0

See Also

- Operator Summary
- Operator Precedence
- Not operator
- BitAnd operator
- BitOr operator
- BitXor operator

Or Operator

Used to perform a logical disjunction on two expressions.

```
result = expression1 Or expression2
```

Arguments

The **Or** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

If either or both expressions evaluate to True, *result* is True. The following table illustrates how *result* is determined:

If *expression1* is	If *expression2* is	The *result* is
True	True	True
True	False	True
False	True	True
False	False	False

See Also

- Operator Summary
- Operator Precedence
- BitOr operator
- And operator
- Not operator
- Xor operator

BitOr Operator

Used to perform a bitwise disjunction on two numeric expressions.

```
result = expression1 BitOr expression2
```

Arguments

The **BitOr** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The **BitOr** 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	If bit in *expression2* is	The *result* is
0	0	0
0	1	1
1	0	1
1	1	1

See Also

- Operator Summary
- Operator Precedence
- Or operator
- BitAnd operator
- BitNot operator
- BitXor operator

Xor Operator

Used to perform a logical exclusion on two expressions.

```
result = expression1 Xor expression2
```

Arguments

The **Xor** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The *result* is determined according to the following table:

If *expression1* is	If *expression2* is	The *result* is
True	True	False
True	False	True
False	True	True
False	False	False

See Also

- Operator Summary
- Operator Precedence
- BitXor operator
- And operator
- Not operator
- Or operator

BitXor Operator

Used to perform a bitwise exclusion on two numeric expressions.

```
result = expression1 BitXor expression2
```

Arguments

The **BitXor** operator syntax has these parts:

result	Any numeric variable.
expression1	Any expression.
expression2	Any expression.

Description

The **BitXor** 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	If bit in *expression2* is	The *result* is
0	0	0
0	1	1
1	0	1
1	1	0

See Also

- Operator Summary
- Operator Precedence
- Xor operator
- BitAnd operator
- BitNot operator
- BitOr operator

Caché Basic Constants

Comparison Constants

These constants are always available.

Predefined Constants

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Date Format Constants

These constants are always available.

Predefined Constants

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. For real numbers, display a date 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.

Date and Time Constants

These constants are always available.

Predefined Constants

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

Node Constants

These constants are always available.

Predefined Constants

Constant	Value	Description
vbUndef	0	Indicates that the referenced variable or node is undefined.
vbHasValue	1	Indicates that the referenced variable or node has a data value.
vbHasArray	2	Indicates that the referenced variable or node has sub-nodes.

String Constants

These constants are always available.

Predefined Constants

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

Constant	Value	Description
vbCr	Chr(13)	Carriage return
vbCrLf	Chr(13) & Chr(10)	Carriage return–linefeed combination
vbFormFeed	Chr(12)	Form feed
vbLf	Chr(10)	Line feed
vbNewLine	Chr(13) & Chr(10) *or* Chr(10)	Platform-specific newline character; whatever is appropriate for the platform
vbNullChar	Chr(0)	Character having the value 0
vbNullString	String having value 0	Not the same as a zero-length string (""); used for calling external procedures
vbTab	Chr(9)	Horizontal tab
vbVerticalTab	Chr(11)	Vertical tab

Caché Basic Objects

Err Object

Contains information about runtime errors.

Description

The **Err** object is an intrinsic object with global scope. There is no need to create an instance of it. The **Err** object contains information about runtime errors and it provides **Raise** and **Clear** methods for generating and clearing runtime errors.

The properties of the **Err** object are set by the generator of an error, either the Caché Basic runtime system in response to an error condition or by the program calling the **Raise** method.

When a runtime errors occurs, the properties of the **Err** object are filled with information that uniquely identifies and describes the error condition.

The **Err** object may be referenced anywhere in the Basic program but will only contain data for the last error.

Properties

These properties can contain a numeric or string literal, or an expression that resolves to a literal.

Number

A number that uniquely defines the error code. Numbers from 1 to 512 are reserved by Caché Basic for indication of runtime errors generated by the system. Numbers from 513 and up are reserved for use by the programmer.

Description

A textual description that describes the nature of the error.

Source

A textual description of the source of the error. This may be the name of the program within which the error occurred or it may be the name of an object. If possible, the location within the code where the error occurred might be included.

Methods

Clear()

This clears the error condition and sets the properties of the object to the empty string. The **Clear** method takes no arguments; the parentheses are optional.

Raise(number [,description [,source]])

The method generates a user-defined exception.

The **Raise** method has the following arguments:

number	The error number.
description	*Optional* — A description of the error.
source	*Optional* — A location associated with the error.

When an exception is generated using the **Raise** method, the properties of the **Err** object are first cleared and then set with the corresponding arguments.

The **Raise** method and the **Err** object are commonly used with the **TRY** and **CATCH** statements.

Examples

The following example issues an error using the **Err.Raise** method. After displaying the error arguments, it uses the **Err.Clear** method to clear these error argument values:

```
Main:
  On Error Goto ErrorHandler
  Println "before the error"
  Err.Raise(100,"Deliberate Error","Main line 3")
  Println "after the error"    ' should not print
ErrorHandler:
  Println "Error1: ", Err.Number, " ", Err.Description," ", Err.Source
  Err.Clear()
  Println "Error2: ", Err.Number, " ", Err.Description," ", Err.Source
```

The following example demonstrates how to use the **Err** object in a function:

```
Println ErrorTest(1)
Println ErrorTest(0)

Function ErrorTest(Arg)
  On Error Goto ErrorHandler
  return 1/Arg
ErrorHandler:
  Println "Error ", Err.Number, " ", Err.Description," ", Err.Source
  Err.Clear()
  return 0
End Function
```

See Also

- TRY statement
- CATCH statement
- On Error function

System Object

The System object provides access to properties and methods of Caché components.

Description

The **System** object is an intrinsic object with global scope. There is no need to create an instance of it.

Through the **System** object programs have access to properties and methods related to the following components: **Activate**, **CSP**, **Encryption**, **Help**, **OBJ**, **SQL** and **Version**.

For detailed information on the **System** object, look at the class definitions in the **%SYSTEM** package.

Examples

The following example demonstrates how to use the **System** object:

```
Println System.Version.GetCompBuildOS()
    ' prints the OS version
    ' for which this Caché Version was built
Println System.Version.GetNumber()      'prints the Version number
Println System.Version.GetVersion()     'prints the Version string
```

Please note that the **System** object in ObjectScript is referenced by **$System** and in Caché Basic by **System**.

Note

The **System** object may be referenced anywhere in the basic program.

Caché Basic General Concepts

Multidimensional Data Model

Introduction to the Multidimensional Data Model in Caché

Rich Multidimensional Data Structure

Caché's high-performance database uses a multidimensional data model that allows efficient and compact storage of data in a rich data structure. With Caché, it is possible to access or update data without performing the complicated and time consuming joins required by relational databases.

Although sometimes described as a "hyper-cube" or "n-dimensional space," a more accurate description of the Caché storage model is a collection of sparse multidimensional arrays called "globals." Data can be stored in a global with any number of subscripts. What's more, subscripts are typeless and hence can be anything – string, integer, floating point, etc. This means one subscript might be an integer, such as 34, while another could be a meaningful name, like "LineItems"– even at the same subscript level.

For example, a stock inventory application that provides information about item, size, color, and pattern might have a structure like this:

```
^Stock(item,size,color,pattern) = quantity
```

Here's some sample data:

```
^Stock("slip dress",4,"blue","floral")=3
```

With this structure, it is very easy to determine if there are any size 4 blue slip dresses with a floral pattern – simply by accessing that data node. If a customer wants a size 4 slip dress and is uncertain about color and pattern, it is easy to display a list of all of those by cycling through all of the data nodes below ^Stock("slip dress",4).

In this example, all the data nodes were of a similar nature (they stored a quantity), and they were all stored at the same subscript level (4 subscripts) with similar subscripts (the 3rd subscript was always text representing a color). However, they do not have to be. Not all data nodes have to have the same number or type of subscripts, and they may contain different types of data.

Here is an example of a more complex global with invoice data that has different types of data stored at different subscript levels:

```
^Invoice(invoice #,"Customer") = Customer information
^Invoice(invoice #,"Date") = Invoice date
^Invoice(invoice #,"Items") = # of Items in the invoice
^Invoice(invoice #,"Items",1,"PartNum") = part number of 1st Item
^Invoice(invoice #,"Items",1,"Quantity") = quantity of 1st Item
^Invoice(invoice #,"Items",1,"Price") = price of 1st Item
^Invoice(invoice #,"Items",2,"PartNum") = part number of 2nd Item
```

Multiple Data Elements per Node

Often only a single data element is stored in a data node, such as a date or quantity, but sometimes it is useful to store multiple data elements together as a single data node. This is particularly useful when there is a set of related data that is often accessed together. It can also improve performance by requiring fewer accesses of the database, especially when networks are involved.

For example, in the previous invoice, each item included a part number, quantity, and price all stored as separate nodes, but they could be stored in a single node:

```
^Invoice(invoice #,"LineItems",item #) = $LB(PartNum,Quantity,Price)
```

To make this simple, Caché supports a list functions which can assemble multiple data elements into a length delimited byte string and later de-assemble them, preserving datatype.

Transaction Processing with a Large Number of Users

Efficient access to data makes the multidimensional model a natural for transaction processing. Caché processes do not have to spend time joining multiple tables, so they run faster.

Logical Locking Promotes High Concurrency

In systems with thousands of users, reducing conflicts between competing processes is critical to providing high performance. One of the biggest conflicts is between transactions wishing to access the same data.

Caché processes do not lock entire pages of data while performing updates. Because transactions require frequent access or changes to small quantities of data, database locking in Caché is done at a logical level. Database conflicts are further reduced by using atomic addition and subtraction operations, which do not require locking. (These operations are particularly useful in incrementing counters used to allocate ID numbers and for modifying statistics counters, both of which are common "hot spots" in a database that would otherwise cause frequent conflicts between competing transactions.)

With Caché, individual transactions run faster, and more transactions can run concurrently.

Multidimensional Model Enables Realistic Description of Data

The multidimensional model is also a natural fit for describing and storing complex data. Developers can create data structures that accurately represent real-world data, thus making it faster to develop applications and easier to maintain them.

Variable Length Data in Sparse Arrays

Because Caché data is inherently of variable length and is stored in sparse arrays, Caché often requires less than half of the space needed by a relational database. In addition to reducing disk requirements, compact data storage enhances performance because more data can be read or written with a single I/O operation, and data can be cached more efficiently.

Declarations and Definitions are Not Required

No declarations, definitions, or allocations of storage are required to directly access or store data in the database, and there is no need to specify the number or type of subscripts or the type or size of data. The multidimensional arrays are inherently typeless, both in their data and subscripts. Global data simply pops into existence as data is inserted with the SET command.

However, to make use of the object access and SQL access of the database, data dictionary information is required. In specifying the data dictionary for objects and SQL, developers have a choice of letting wizards automatically select the multidimensional data structure best suited to their data, or they can directly specify the mapping.

Namespaces

In Caché, data and ObjectScript code are stored in disk files with the name CACHE.DAT (only one per directory). Each such file contains numerous "globals" (multidimensional arrays). Within a file, each global name must be unique, but different files may contain the same global name. These files may be loosely thought of as databases.

Rather than specifying which CACHE.DAT file to use, each Caché process uses a "namespace" to access data. A namespace is a logical map that maps the names of multidimensional global arrays and routine code to CACHE.DAT files, including the Data Server and directory name for that file. If a file is moved from one disk drive or computer to another, the namespace map is changed.

Usually a namespace specifies sharing of certain system information with other namespaces, and the rest of the namespace's data is in a single CACHE.DAT used only by that namespace. However, this is a flexible structure that allows arbitrary mapping, and it is not unusual for a namespace to map the contents of several CACHE.DAT files

Reserved words

A list of Caché Basic reserved words.

```
ABS | AND | AS | ASC | ATN |
BITAND | BITEQV | BITIMP | BITNOT | BITOR |
BITXOR | BYREF | BYVAL |
CALL | CASE | CATCH | CHR | CONST | CONTINUE | COS |
DATE | DATEADD | DATEDIFF | DATEPART |
DAY | DEBUG | DEFAULT | DIM | DO |
EACH | ELSE | ELSEIF | END |
EQV | ERR | ERROR | EXISTS |
EXIT | EXP | EXPLICIT |
FALSE | FINALLY | FIX | FOR | FUNCTION |
GOTO |
HEX | HOUR |
IF | IMP | IN | INCREMENT |
INPUT | INSTR | INSTRREV | INT |
IS | ISOBJECT |
JOIN |
LCASE | LEFT | LEN | LET |
LIST | LISTBUILD | LISTEXISTS | LISTFIND | LISTGET |
LISTLENGTH | LOCK | LOG | LOOP | LTRIM |
ME | MID | MINUTE | MOD | MONTH | MONTHNAME |
NEW | NEXT | NOT | NOTHING | NOW |
OBJECT | OCT | ON | OPEN | OPENID | OPTION | OR |
PIECE | PRIVATE | PUBLIC |
RANDOMIZE | REM | REPLACE | RETURN |
RIGHT | RND | ROUND | RTRIM |
SECOND | SELECT | SET | SGN | SIN | SPACE |
SPLIT | SQR | STRCOMP | STRING | STRREVERSE | SUB | SYSTEM |
TAN | TCOMMIT | THEN | THROW | TIMER | TRAVERSE |
TRIM | TROLLBACK | TRUE | TRY | TSTART |
UCASE | UNLOCK | UNTIL |
VBBACKWARD | VBCR | VBCRLF | VBFF | VBFORMFEED | VBFORWARD |
VBFRIDAY | VBHASARRAY | VBHASVALUE | VBLF | VBMONDAY | VBNC |
VBNEWLINE | VBNL | VBNULLCHAR | VBSUNDAY | VBTAB | VBTHURSDAY |
VBTOEXTERNAL | VBTOINTERNAL | VBTUESDAY | VBUNDEF |
VBUSESYSTEM | VBVT | VBVERTICALTAB | VBWEDNESDAY |
WEEKDAY | WEEKDAYNAME | WEND | WHILE |
XOR |
YEAR
```

Description

Within Caché Basic certain words are *reserved*. You cannot use a Basic reserved word as a Basic identifier (such as the name of a variable).