Using Caché Basic

Version 2018.1
2024-05-02

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

Using Caché Basic
Caché Version 2018.1 2024-05-02

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

List of Tables

About This Book

This book explains the various elements of the Caché Basic programming language, and how to use this language as part of Caché.

Its chapters are:

- Introduction to Basic
- Lexical Structure
- Data Types and Values
- Variables
- Operators and Expressions
- Statements
- User-defined Basic Code
- Arrays
- Using Objects
- Error Handling
- Frequently Asked Questions About Caché Basic

There is also a detailed Table of Contents.

The following documents provide information about Caché Basic and related concepts:

- The *Caché Basic Reference* provides descriptions of all of the commands, functions, and operators supported by Caché Basic.
- *Using Caché Objects* provides a description on how Caché objects work.

For general information, see *Using InterSystems Documentation*.

1

Introduction to Basic

Caché Basic is an easy-to-learn, object-oriented scripting language designed for rapid development of powerful, web-based database applications. Developers familiar with Visual Basic will find it easy to develop applications using Caché Basic.

Caché Basic is seamlessly integrated with Caché objects and supports object, relational, and multidimensional access to data.

Caché Basic is completely compatible and interoperable with the other Caché scripting language, ObjectScript. Like ObjectScript code, Basic code is compiled into object code, which is executed by a highly optimized virtual machine.

The definition of the Caché Basic is based on VBScript, enhanced by language elements necessary to communicate with the database layer of Caché. The following table summarizes some of the key differences between VBScript and Caché Basic:

Table 1–1: Comparison of VBScript and Caché Basic

Feature	VBScript	Caché Basic
Basic Syntax	Yes	Yes
Object Support	Yes	Yes
Multidimensional Arrays	No	Yes
Additional String Operations	No	Yes
User Interface Components	Yes	No
Portability	No	Yes

The main thing to keep in mind when comparing VBScript and Caché Basic is that they are designed for different tasks: VBScript is user interface automation language designed for use within client-based applications, such as browsers. Caché Basic is a server-based language designed to efficiently run portable business logic for Web-based, database applications.

2

Lexical Structure

Lexical structure is the set of fundamental rules that govern the behavior of a programming language. This chapter describes the lexical structure of Caché Basic.

```
a.foo = 10
A.foo = 20
PrintLn "a = " & a.foo
PrintLn "A = " & A.foo
```

2.1 Case Sensitivity

Statement, function and operator names are not case-sensitive in Basic:

```
PrintLn "Hello"
PRINTLN "Hello"
println "Hello"
```

Variable names are case-sensitive:

```
a = 10
A = 20
PrintLn "a = " & a
PrintLn "A = " & A
```

Object identifiers (class, method, and property names) are not case-sensitive, but must be locally unique as if they were case-sensitive (in other words, you cannot have a Person and a PERSON class):

```
person = OpenId Sample.Person(1)
PrintLn person.Name
' PrintLn person.name  ' Error
```

2.2 White Space and Line Continuation

Basic ignores spaces and tabs that appear between tokens. A token is either a function name, variable name, number, or a keyword. White space cannot appear *within* a token.

A newline indicates the end of a Basic code statement. The underscore character (_) is the line continuation indicator. To continue a statement on the next line, end the first line with a space character followed by an underscore. The underscore *must* be the last character of the line; a space or comment is not permitted after the line continuation character.

2.3 Comments

Comments in Basic are line oriented and start with either the **Rem** statement or the apostrophe ('). The comment is always until the end of the line and everything within the comment is ignored:

Basic example:

```
' This is a comment
```

2.4 Literals

Literals appear within the program and represent a data value. A literal can be a constant (`vbMonday`).

The following example prints a variety of literals :

```
PrintLn "This is a String" 'String literal
PrintLn 1234 ' Integer literal
PrintLn 123.4 ' Numeric literal
PrintLn true
PrintLn vbMonday
```

2.5 Identifiers and Variables

An identifier is a name and can be used to name variables, functions, classes, and so on.

A variable is a named storage location that can contain data that can be modified during program execution. Each variable has a name that uniquely identifies it within its level of scope. There are three types of variables: local variables, process-private global variables, and global variables. For detailed definitions of these three types of variables, refer to the "Variables" chapter of *Using Caché ObjectScript*.

Variable names are case-sensitive. Variable names may be of any length, but must be unique within the first 31 characters (exclusive of prefix characters). Variables can have multiple levels of subscripts. For details on variable naming conventions in Caché Basic, refer to the "Variables" chapter of this manual.

2.6 Labels

Within Caché Basic code you can define a label as follows:

Basic example:

```
labelOne:
    ' some code
labelTwo:
    ' some more code
```

A label is a valid identifier followed by a colon (:). Label names are case-sensitive. A label name can begin with either a letter or a percent (%) character; the other characters in the label name must be letters or numbers. A label can be of any length, but only the first 31 characters are meaningful. A label can have the same name as a variable or a reserved word, though this is not recommended.

A label must be the first item on a program line. A label does not have to begin at column 1, although this is generally recommended. You can specify more than one label on the same program line, separating these labels with a blank space.

Following all labels on a program line, you can specify one or more commands on the same line; multiple commands on the same program line are separated by a colon character. Following all labels and commands on a program line, you can specify a comment. For example, the following is a valid line of Caché Basic code:

```
label1: label2: Set x="text" : Println x ' my comment
```

Labels are used by the Goto statement and the On Error Goto statement, which are documented in the *Caché Basic Reference*. Refer to the Error Handling chapter of this manual for additional information on the **On Error Goto** statement.

2.7 Reserved Words

Within Caché Basic there are a number of reserved words that cannot be used as local variable names or function names.

You can find the list of reserved words in the Caché Basic Reference.

3

Data Types and Values

This chapter gives a brief overview of the various data types in Basic.

3.1 Numbers

Numbers are a sequence of digits which may contain a decimal point. Basic also supports exponential notation to represent numbers.

Some examples of numbers:

```
PrintLn 1
PrintLn 1.1
PrintLn 1E10
```

3.2 Strings

Strings in Basic are surrounded by double quotes (") and represent a sequence of letters, digits, and punctuation. The minimum string length is zero. By default, the maximum string length is 32K characters. If you have *long strings* enabled in your installation, the maximum string length is 3,641,144 characters. For further details, see "Enabling Long String Operations" in the *Caché Programming Orientation Guide*.

Some examples of strings:

```
PrintLn "Hello"
PrintLn ""
PrintLn "Goodbye"
```

To include a double quote character as a literal within a string, use a pair of double quote characters, as shown in the following example:

```
PrintLn "Hello ""World"""
```

You can concatenate strings using the concatenation operator (&):

```
PrintLn "Hello" & " " & "Goodbye"
```

Caché Basic includes a number of specialized string operations. These are listed in the Caché Basic Reference.

3.3 Persistent Multidimensional Arrays (Globals)

A Global is a sparse multidimensional database array. Data can be stored in a global with any number of subscripts. Subscripts in Caché are typeless.

A global is not different from any other type of array, with the exception that the global variable name starts with a caret (^).

Some examples of globals. First, run this example to set the global ^*x*:

```
^x = 10
PrintLn "^x has been defined"
```

Now, run this example to examine the value of ^*x*:

```
PrintLn "The value of ^x is: " & ^x
```

Similarly, since globals are arrays, you can set the value of an array reference:

```
^y("string",1) = 2
^y("string","a") = "b"
PrintLn "Values in ^y are " & ^y("string",1) & " and " & ^y("string","a")
```

For more information on arrays, see the "Arrays" chapter in this document. For more information on globals, see the *Using Caché Globals* document.

3.4 Null

The **Null** keyword is not supported in this version of Basic.

3.5 Undefined

Variables in Caché Basic do not need to be declared or defined. By simply assigning a value to a variable it is defined, but until this first assignment all references to this variable are undefined. You can use the Exists function to determine if a variable is defined or undefined:

```
PrintLn "Does x exist? " & Exists(x)
x = 10
PrintLn "How about now? " & Exists(x)
```

3.6 Dates

A date value represents a valid date within the range 31 December, 1840 through 31 December, 9999.

You can use the Now function to get the current date and time:

```
today = Now()
PrintLn "Today is: " & today
```

Similarly, you can use the Date and Time functions to get the either the current date or time, respectively:

```
PrintLn "Date is: " & Date()
PrintLn "Time is: " & Time()
```

There are a number of functions for extracting component information from date values. These include Day, Hour, Minute, Month, Second, Weekday, and Year:

```
today = Now()
PrintLn "Today is: " & today
PrintLn "Year: " & Year(today)
PrintLn "Month: " & Month(today)
PrintLn "Day: " & Day(today)
PrintLn "Weekday: " & Weekday(today)
PrintLn "Hour: " & Hour(today)
PrintLn "Minute: " & Minute(today)
PrintLn "Second: " & Second(today)
```

3.7 Objects

An object value refers to a instance of an in-memory object. You can assign an object value to any local variable:

```
person = New Sample.Person()
PrintLn person
```

You can refer to the methods and properties of an object using dot syntax:

Basic example:

```
person.Name = "El Vez"
```

You can determine if a variable contains an object using the IsObject function:

```
str = "A string"
person = New Sample.Person()
tStatement = New %SQL.Statement()

PrintLn "Is string an object? " & IsObject(str)
PrintLn "Is person an object? " & IsObject(person)
PrintLn "Is tStatement an object? " & IsObject(tStatement)
```

You cannot assign an object value to a global. Doing so will result in a runtime error.

Assigning an object value to a variable (or object property) has the side effect of incrementing the object's internal reference count. When the number of references to an object reaches 0, Caché will automatically destroy the object (invoke its **%OnClose** method and remove it from memory).

4

Variables

A variable is the name of a location in which a runtime value can be stored. There are three types of variables: local variables, process-private global variables, and global variables.

4.1 Variable Names

Each variable in Caché Basic has a name that uniquely identifies it within its level of scope. Variable names are case-sensitive. Variable names may be of any length, but must be unique within the first 31 characters (exclusive of prefix characters). Variables can have multiple levels of subscripts.

- Local variable names must begin with an alphabetic or percent character (%). The second and subsequent characters can be alphabetic characters, numbers, or the underscore (_) character. The scope of local variables is limited to the current process. Local variables are accessible from any namespace. The following are all valid local variable names:

```
a = 10
A = 20
a(1,3) = "subscripted"
%A = "percent and letter"
%5 = "percent and number"
aa = "letters"
a5 = "letter and number"
aBcDeFgHiJkLmNoPqRsTuVwXyZ = "alphabet"
a_e_i_o_u_ = "vowels"
PrintLn "a = " & a
PrintLn "A = " & A
PrintLn "a(1,3) = " & a(1,3)
PrintLn "%A = " & %A
PrintLn "%5 = " & %5
PrintLn "aa = " & aa
PrintLn "a5 = " & a5
PrintLn "aBc... = " & aBcDeFgHiJkLmNoPqRsTuVwXyZ
PrintLn "a_e_i_o_u_ = " & a_e_i_o_u_
```

  Note that local variable names can include underscore (_) characters, but cannot include dot (.) characters.

- Process-private globals in Caché Basic can take either of the following syntactical forms: `^||name` or `^|"^"|name`. Other process-private global syntactical forms are not supported in Caché Basic. The first character of a process-private global name can be either an alphabetic or percent character (%). The second and subsequent characters can be alphabetic characters, numbers, or dot (.) characters. A dot character (period) may not be the first or last character of the name.

The scope of process-private globals is limited to the current process. Process-private globals are accessible from any namespace. The following are all valid process-private global names:

```
^||a = 10
^|"^"|A = 20
^||a(1,3) = "subscripted"
^||%A = "percent and letter"
^||%5 = "percent and number"
^||aa = "letters"
^||a5 = "letter and number"
^||aBcDeFgHiJkLmNoPqRsTuVwXyZ = "alphabet"
^||a.e.i.o.u = "vowels"
PrintLn "^||a = " & ^||a
PrintLn "^|"^"|A = " & ^|"^"|A
PrintLn "^||a(1,3) = " & ^||a(1,3)
PrintLn "^||%A = " & ^||%A
PrintLn "^||%5 = " & ^||%5
PrintLn "^||aa = " & ^||aa
PrintLn "^||a5 = " & ^||a5
PrintLn "^||aBc... = " & ^||aBcDeFgHiJkLmNoPqRsTuVwXyZ
PrintLn "a.e.i.o.u = " & ^||a.e.i.o.u
```

Note that process-private global names can include dot (.) characters, but cannot include underscore (_) characters.

- Globals in Caché Basic can take any of the following syntactical forms: `^name`, `^|""|name`, or `^|"namespace"|name`. Global syntactical forms using square brackets are not supported in Caché Basic. The first character of a global name can be either an alphabetic or percent character (%). The second and subsequent characters can be alphabetic characters, numbers, or dot (.) characters. A dot character (period) may not be the first or last character of the name. The scope of globals is not limited to the process that created it. A global is stored in the database and is therefore visible to all processes. A global persists after the process that created it terminates. A global is specific to a namespace, and can only be access from that namespace, or by referencing that namespace. The following are all valid global names:

  Basic example:

  ```
  ^a = 10
  ^|"samples"|A = 20
  ^a(1,3) = "subscripted"
  ^%A = "percent and letter"
  ^%5 = "percent and number"
  ^|""|aa = "letters"
  ^a5 = "letter and number"
  ^aBcDeFgHiJkLmNoPqRsTuVwXyZ = "alphabet"
  ^a.e.i.o.u = "vowels"
  PrintLn "^a = " & ^a
  PrintLn "^A = " & ^|"samples"|A
  PrintLn "^a(1,3) = " & ^a(1,3)
  PrintLn "^%A = " & ^%A
  PrintLn "^%5 = " & ^%5
  PrintLn "^aa = " & ^|""|aa
  PrintLn "^a5 = " & ^a5
  PrintLn "^aBc... = " & ^aBcDeFgHiJkLmNoPqRsTuVwXyZ
  PrintLn "a.e.i.o.u = " & ^a.e.i.o.u
  ```

  Note that global names can include dot (.) characters, but cannot include underscore (_) characters.

For further details on percent variables and system names, see "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*. For further details on using globals as subscript arrays, see *Using Caché Globals*.

4.2 Variable Typing

Variables in Caché Basic are, like in ObjectScript, JavaScript and VBScript, untyped. This means that you can assign a string value to a variable and, later on, a numeric value to the same variable. Caché automatically converts the value of variables to other types, based on the context in which it is used.

Conversion amongst the various types are carried out using the same rules as ObjectScript.

Table 4–1: Caché Basic Type Conversion Rules

From	To	Rules
Number	Object	Not allowed.
Number	String	The number is converted to a string of numeric characters that corresponds to the number's value: `22` becomes `"22"`.
Object	Number	The object reference is converted to an integer whose value is the object instance number.
Object	String	The object reference is converted to a string in the form: "oref@classname" where *oref* is the object instance number and *classname* is the type of object.
String	Number	The string is parsed left-to-right until a non-numeric character is encountered. The leading numeric characters are converted to the corresponding numeric value. If there are no leading numeric characters, the string is converted to `0`. For example, `"123"` converts to `123`, `"123abc"` converts to `123`, and `"abc123"` converts to `0`.
String	Object	Not allowed.

4.3 Variable Declaration

Declaring variables in Caché Basic is not required. If you want to take advantage of detecting undeclared variables by the compiler you can use the **Option Explicit** clause at the beginning of your program. In that case you must declare all variables using the **Dim** statement:

Basic example:

```
Option Explicit
Dim a
```

4.4 Variable Scope

Variables used in Caché Basic are visible within the routine, unless a procedure or function explicitly declares the variable using the **Dim** statement.

4.5 Default Variable Values

When initially created, a variable is *Empty*, which indicates that no value has been assigned to the variable. Empty variables are 0 in a numeric context, or zero-length in a string context. Unlike ObjectScript, invoking an undefined variable does not result in an error. If no value has been assigned to a variable, invoking that variable returns the empty (null) string.

4.6 Maximum Number of Variables

A Caché Basic routine can contain a maximum of 32,759 private variables, and a maximum of 65,280 public variables. Exceeding these limits results in a compile error.

5

Operators and Expressions

This chapter provides an overview of expressions and operators within Caché Basic.

5.1 Expressions

An expression is a combination of keywords, operators, variables, and constants that yield a string, number, or object value. An expression can perform a calculation, manipulate characters, or test data.

5.2 Operators

Caché Basic includes a variety of built-in operators.

5.2.1 Operator Precedence

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
Exponentiation (^)	Equality (=)	Not
Negation (-)	Inequality (<>)	And
Multiplication and division (*, /)	Less than (<)	Or
Integer division (\)	Greater than (>)	Xor
Modulus arithmetic (Mod)	Less than or equal to (<=)	Eqv
Addition and subtraction (+, -)	Greater than or equal to (>=)	Imp
String concatenation (&)	Is	&

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.

5.3 Comparison Operators

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.

6

Statements

Programs consist of a series of statements. Simple statements assign or retrieve values to and from variables, while complex statements like **If** and loops support program flow or provide powerful functionality using objects.

6.1 Expression Statements

Expression statements are used to assign values to variables or apply an operation. For example:

Basic example:

```
msg = "Hello"
result = obj.Method()
Print x
```

An assignment expression sets the value of a variable, array element, or object property:

Basic example:

```
var = 1
array(1) = 2
obj.Property = 3
objarray(1).Property = 4
```

The end of a statement is normally marked by a carriage return. You can, however, place multiple statements on the same line by placing the colon (:) character between them:

```
a = 1 : b = 2 : c = 3
Print a : Print " " : Print b : Print " " : Print c
```

6.2 Call

The **Call** statement lets you call a function or method from within Basic:

Basic example:

```
Call obj.Method(a,b)
```

Within a **Call** statement, the actual “Call” token is completely optional and can be omitted:

Basic example:

```
obj.Method(a,b)
```

By default, all function arguments are passed by reference within Basic. You can override this default on a argument-by-argument basis using the **ByRef** and **ByVal** directives:

Basic example:

```
obj.Method(ByRef a, ByVal b)
```

6.3 If

The **If** statement provides the ability to execute statements conditionally. Within Caché Basic, an **If** statement takes the following form:

Basic example:

```
If (expression) Then
    statement()
End If
```

If *expression* evaluates to true, the body of the **If** statement is executed. The body of the **If** statement consists of a block of one or more statements between a **Then** and **End If** tokens.

There is also a simpler, single-line form of the **If** statement:

Basic example:

```
If (expression) Then statement()
```

In the single-line form, there is no **End If** token; the body of the **If** statement follows the **If** token and continues to the end of the line. Note that you can include multiple statements by separating them with colon (:) characters:

Basic example:

```
If (expression) Then statement1() : statement2()
```

An **If** statement make contain an **Else** clause:

Basic example:

```
If (expression) Then
    statement1()
Else
    statement2()
End If
```

The statements within the **Else** block are executed if *expression* evaluates to False.

6.4 ElseIf

ElseIf is an optional statement which is part of the closest **If** statement. The **ElseIf** statement allows an **If** statement to sequentially test a set of expressions:

Basic example:

```
If (expression) Then
    statement1()
ElseIf (expression2) Then
    statement2()
ElseIf (expression3) Then
    statement3()
Else
    statement3()
End If
```

The statements within the first **If** or **ElseIf** block whose expression is true are executed. If none are true, then the **Else** block is executed.

6.5 For

The **For** statement is used to repeat a sequence of statements.

6.5.1 For...Next

The **For...Next** loop uses a counter to execute a block of statements a specified number of times:

```
For i = 1 To 10
    PrintLn i
Next
```

In this case, *i* will be initialized to 1 and the body of the **For...Next** loop will be executed until *i* reached 10. The value of *i* is incremented by 1 after each iteration.

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify. To decrease the counter variable, use a negative **Step** value. You must specify an end value that is less than the start value:

```
For i = 10 To 1 Step -1
    PrintLn i
Next
PrintLn "Blast Off!"
```

If you like (say for legibility), you can place the counter variable after the **Next** token:

```
For i = 1 To 10
    PrintLn i
Next i
```

Using the wrong variable after the **Next** token results in a syntax error.

6.5.2 For Each...Next

The **For Each...Next** repeats a group of statements for each item in a collection or each element of an array. A **For Each...Next** loop is similar to a **For...Next** loop. Instead of repeating the statements a specified number of times, a **For Each...Next** loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you don't know how many elements are in a collection.

For example, the following creates an array of ten items and then uses a **For Each...Next** loop to display the subscript of each item:

```
' Create a simple array
For i = 1 To 10
    array(i) = i * i
Next

' Now display the subscripts and contents of the array
For Each key in array
    PrintLn key & ": " & array(key)
Next
```

With a multidimensional array, this gets more interesting:

```
' Create a multidimensional array
For i = 1 To 5
    For j = 1 To 5
        array(i, j) = i * j
    Next j
Next i

' Now display the subscripts and contents of the array
For Each key1 in array
    For Each key2 in array(key1)
        PrintLn key1 & "," & key2 & ": " & array(key1,key2)
    Next key2
Next key1
```

6.6 While

You can use a **While** statement to run a block of statements an indefinite number of times:

Basic example:

```
While (expression)
    statement()
Wend
```

The statements are repeated as long as *expression* is True. The end of the **While** block is marked by the **Wend** token.

For example, the following code executes its **While** block 9 times:

```
x = 1
While (x < 10)
    PrintLn x
    x = x + 1
Wend
```

6.7 Do...While

You can use a **Do...While** statement to run a block of statements an indefinite number of times:

Basic example:

```
Do
    statement()
Loop While (expression)
```

The statements are repeated as long as *expression* is True. The **Do...While** block is contained within the **Do** and **Loop** tokens.

The **Do...While** statement differs from the **While** statement in that it executes its contents *before* evaluating its loop expression.

6.8 Select ... Case

The Select...Case statement lets a program choose one of several execution paths based on evaluating an expression. It is a convenient alternative to a long series of **ElseIf** statements.

A **Select...Case** statement takes the following form:

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

The expression *testexpression* is evaluated and its result is then compared with each of the *expressionlist* values associated with each **Case** clause. The code block associated with the first matching **Case** clause is then executed. If there is no match, then the **Case Else** block (if present) is executed.

For example:

```
x = 2
Select Case x
    Case 1
        PrintLn "First case"
    Case 2
        PrintLn "Second case"
    Case Else
        PrintLn "Unknown"
End Select
```

There is also a simpler form of **Case** that can be used as an in-line expression:

Basic example:

```
Case(expr, val1:ret1, val2:ret2)
```

In this case (pun intended), the expression *expr* is first evaluated, and then its value is compared with each of a series of values (*val1*, *val2*, in this example). The value associated with the first matching value is returned.

For example:

```
x = 2
PrintLn Case(x,1:"hello",2:"goodbye")
```

6.9 Dim

To declare local variables Caché Basic supports the **Dim** statement. There is no need to declare variables in Caché Basic and the declaration is optional unless you include the **Option Explicit** clause in your program.

6.10 Return

You can return a value from a method using the **Return** statement:

Class Member

```
Method Add(a As %Integer, b As %Integer) As %Integer [language = basic]
{
    Return a + b
}
```

Note that it is a syntax error to use the **Return** statement within a method that does not return a value.

7

User-defined Basic Code

A typical application consists of some number of user-defined methods and functions which contain the actual application logic. These exist as procedures and code within them is known as "procedure-level code"; code outside of the procedures' code blocks is known as "script-level code."

7.1 Basic Methods

Within a Caché Basic application, the user-defined business logic (the code for the application) is typically contained within methods defined within classes:

Class Definition

```
Class MyApp.MyClass [language = basic]
{

Method MyMethod() As %Integer
{
    a = 22
    Return a
}

}
```

Note that the primary language for this class is specified (using the language keyword) as "Basic". This means that all methods will be implemented using Basic. You can override this on a method-by-method basis and it is possible to mix methods using different languages together within the same class.

7.1.1 ##super

In a Basic method, you can use the syntax ##super to invoke a method by the same name, inherited from a superclass. (For a general discussion of ##super, see "##super syntax" in "Object-specific ObjectScript Features" in *Using Caché Objects*.)

If a Basic method uses ##super, and if multiple superclasses define the same method, the system invokes the method in the class that appears first (leftmost) in the inheritance list, regardless of the value of the Inheritance keyword.

7.1.2 Calling a Method

You can call a Basic method from Basic:

Basic example:

```
x = obj.MyMethod()
```

or from ObjectScript:

ObjectScript

```
 Set x = obj.MyMethod()
```

7.2 Basic Routines and Functions

You can create a Basic *routine* (similar to an ObjectScript routine but implemented using Basic) and define one or more functions within it:

Basic example:

```
 'MyRoutine.BAS

Function Add(a,b)
    Return a + b
End Function
```

You can call this function from Basic (from outside the MyRoutine.BAS routine):

Basic example:

```
x = Add@MyRoutine(2,3)
```

or from ObjectScript:

ObjectScript

```
 Set x = $$Add^MyRoutine(2,3)
```

7.3 Calling ObjectScript Functions

Should you need to, it is possible to call ObjectScript functions from Basic.

Note that you can call object methods implemented in ObjectScript from Basic and vice versa using normal object syntax.

7.3.1 ObjectScript Extrinsic Functions

Should you need to call an ObjectScript user-defined (extrinsic) function from Basic, you can do this using a special syntax:

Basic example:

```
val = Func@Routine(args)
```

Where *Func* is the name of the function and *Routine* is the name of the routine containing the function.

For example, suppose you have an ObjectScript routine, Math.INT:

ObjectScript

```
  ; Math.INT
Add(a,b) PUBLIC {
    Quit a + b
}
```

You can call this function from Caché Basic as follows:

Basic example:

```
val = Add@Math(a,b)
```

This is equivalent to the following ObjectScript code:

ObjectScript

```
 Set val = $$Add^Math(.a,.b)
```

Note the following differences between Basic and ObjectScript:

1. There is no **Set** command needed in Basic.
2. You do not need to use $$ in Basic for functions that return values.
3. Basic uses the "@" character as a delimiter between the function and routine names.
4. By default, all function arguments are passed by reference in Basic.

   If you want to pass arguments by value in Basic, you need to use the **ByVal** directive:

   ### Basic example:

   ```
   val = Add@Math(ByVal a, ByVal b)
   ```

7.3.2 ObjectScript Intrinsic Functions

From Basic, you cannot directly call built-in (intrinsic) ObjectScript functions, such as those listed in the "ObjectScript Functions" section of the *Caché ObjectScript Reference*. You can, however, create a simple ObjectScript function or method that invokes the desired function and call that from Basic.

For example, the following ObjectScript function invokes $ZCOS to get the cosine of a specified value:

ObjectScript

```
  ; MyMathUtils.INT
COS(arg) PUBLIC {
 ; Return the cosine of the argument
    Quit $ZCOS(arg)
}
```

You can call this from Caché Basic as follows:

Basic example:

```
PrintLn "The cosine is " & COS@MyMathUtils(x)
```

8

Arrays

Caché Basic includes much more powerful support for arrays than other versions of Basic. Specifically, with Caché Basic you can:

- Create sparse, multidimensional arrays.
- Use any combination of integer, string, and numeric values are array subscripts. Such arrays are automatically sorted by subscript value; you never need to invoke a sort function.
- Use arrays without having to declare their size and dimensions. You do not need to **Dim** or **Redim** arrays in Caché Basic.
- Use a number of built-in commands for manipulating and traversing arrays.

In addition, you can create and work with persistent multidimensional arrays (globals) that are automatically stored within the database. For more information, refer to Using Caché Globals.

8.1 Defining Arrays

You do not need to declare array variables in any special way. To create an array, you simply place values into it:

```
' Create an array
arr(1) = "First element"
arr(2) = "Second element"
arr(3) = "Third element"

' Display the contents of the array
For i = 1 To 3
    PrintLn i & ": " & arr(i)
Next
```

Each array element (node) can contain up to 32K of data.

To create a multidimensional array, simply use more subscripts:

```
' Create a multidimensional array
arr(1,1) = "X" : arr(1,2) = "O" : arr(1,3) = "X"
arr(2,1) = "X" : arr(2,2) = "X" : arr(2,3) = "O"
arr(3,1) = "O" : arr(3,2) = "O" : arr(3,3) = "X"

' Display the contents of the array
For row = 1 To 3
    For col = 1 To 3
        Print arr(row,col) & " "
    Next col
    PrintLn
Next row
```

You can use strings as array subscripts:

```
' Create an associative array
notes("Jack") = "Fell down; Broke crown"
notes("Jill") = "Tumbled after"

' Display the contents of the array
PrintLn "Jack: " & notes("Jack")
PrintLn "Jill: " & notes("Jill")
```

8.1.1 Existence Testing

You can test if an array node (or any variable) is defined (has a value) or has sub-nodes using the Exists function. The **Exists** function returns one of the following possible values:

Table 8–1: Exists Return Values

Constant	Value	Meaning
`vbUndefined`	0	Variable is undefined.
`vbHasValue`	1	Variable has a value.
`vbHasArray`	2	Variable is not defined but has subnodes.
`vbHasValue` AND `vbHasArray`	3	Variable is defined *and* has subnodes.

For example:

```
' Initialize
var = "Variable"
arr(1) = "Array node with subnode"
arr(1,1) = "Array node"

' Test for existence
PrintLn "Exists(a)        " & Exists(a)
PrintLn "Exists(var)      " & Exists(var)
PrintLn "Exists(arr)      " & Exists(arr)
PrintLn "Exists(arr(1))   " & Exists(arr(1))
PrintLn "Exists(arr(1,1)) " & Exists(arr(1,1))
```

8.1.2 Deleting Arrays

You can delete an array (its subscripts and values) using the Erase command:

```
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
```

8.2 Traversing Arrays

There are a number of ways to traverse an array to find what subscripts and values it contains.

8.2.1 Traversing with a For...Next Loop

If an array is subscripted by integer values and you know how many elements it contains, you can use a simple For...Next loop to list its contents:

```
' Create an array
For i = 1 To 10
    arr(i) = "Item " & i
Next i

' Display the contents of the array
For i = 1 To 10
    PrintLn i & ": " & arr(i)
Next
```

8.2.2 Traversing with a For Each...Next Loop

You can use a For Each...Next loop to list all the subscripts for an array:

```
' Create an array
food("Apple") = "Fruit"
food("Banana") = "Fruit"
food("Asparagus") = "Vegetable"
food("Broccoli") = "Vegetable"

' Display the array subscripts
For Each key In food
    PrintLn key
Next
```

8.2.3 The Traverse Function

You can use the Traverse function to find the next (or previous) subscript at any point within an array. (**Traverse** is the Basic equivalent of the ObjectScript $ORDER function).

The **Traverse** function takes an array reference and returns the next (or previous) subscript at the specified subscript level. The specified subscript level is simply the last subscript listed in the array reference. For example:

```
' Define an array
arr("a",1) = "a1"
arr("a",2) = "a2"
arr("b",1) = "b1"
arr("b",2) = "b2"

PrintLn Traverse(arr(""))
PrintLn Traverse(arr("a"))
PrintLn Traverse(arr("b"))

PrintLn Traverse(arr("a",""))
PrintLn Traverse(arr("a",1))
```

We can use **Traverse** in a While loop to find all of the subscripts of the *food* array from the previous section:

```
' Create an array
food("Apple") = "Fruit"
food("Banana") = "Fruit"
food("Asparagus") = "Vegetable"
food("Broccoli") = "Vegetable"

' Display the array subscripts
key = Traverse(food("")) ' find first subscript
While (key <> "")
    PrintLn key
    key = Traverse(food(key)) ' find next subscript
Wend
```

To find previous nodes with **Traverse**, use the optional direction flag as a second argument:

```
' Create an array
food("Apple") = "Fruit"
food("Banana") = "Fruit"
food("Asparagus") = "Vegetable"
food("Broccoli") = "Vegetable"

' Display the array subscripts backwards
key = Traverse(food(""),-1) ' find last subscript
While (key <> "")
    PrintLn key
    key = Traverse(food(key),-1) ' find previous subscript
Wend
```

9

Using Objects

Caché includes a full-featured, next-generation object database specifically designed to meet the needs of complex, transaction oriented applications.

The Caché Object Model includes the following features:

- *Classes* — You can define classes that represent the state (data) and behavior (code) of your application components. Classes are used to create instances of objects as both runtime components and as items stored within the database.
- *Properties* — Classes can include properties, which specify the data associated with each object instance. Properties can be simple literals (such as strings or integers), user-defined types (defined using data type classes), complex (or embedded) objects, collections, or references to other objects.
- *Relationships* — Classes can define how instances of objects are related to one another. The system automatically provides navigational methods for relationships as well as referential integrity within the database.
- *Methods* — Classes can define behavior by means of methods: executable code associated with an object. Object methods run within a Caché server process (though they can be invoked from a remote client). Object methods can be scripted using ObjectScript, SQL, or they can be generated using method generators, which are code that automatically creates customized methods according to user-defined rules.
- *Inheritance* — By deriving new classes from existing ones, you can reuse previously written code as well as create specialized versions of classes.
- *Polymorphism* — Caché supports complete object polymorphism. This means that applications can use a well-defined interface (a set of methods and properties provided by a superclass) and the system will automatically invoke the correct interface implementation based on the type of each object. This makes it much easier to develop flexible database applications.
- *Swizzling* (also known as “lazy loading”) — Caché automatically swizzles (brings into memory from disk) any related persistent objects when they are referenced from other objects. This greatly simplifies working with complex data models.

For detailed information on how to use Objects in Caché, refer to Using Caché Objects.

9.1 Defining Classes

An object is an instance of a specific class. Within Caché you can define classes in several ways. For information on how to define classes refer to the following sections in *Using Caché Objects*:

- Defining and Compiling Classes

- Introduction to Caché Objects"

9.2 Creating Object Instances

You can create an object instance by either creating a new instance using the **New** command or you can open a persistent object previously stored in the database using the **OpenId** command.

9.2.1 The New Command

The **New** command creates and returns a new instance of the specified class. For example:

```
person = New Sample.Person()
person.Name = "El Vez"
PrintLn person.Name
```

The name of the class (with an optional package name) follows the **New** command. There are parentheses following the class name. If you place an argument within these parentheses, then it will be passed to the constructor (**%OnNew** method) for the object.

The **New** command has exactly the same behavior as invoking the **%New** method in ObjectScript.

9.2.2 The OpenId Command

The **OpenId** command finds, opens, and returns an instance of a persistent class that is stored within the database using its unique object identifier value. For example:

```
person = OpenId Sample.Person(22)
PrintLn "Name: " & person.Name
PrintLn "Age: " & person.Age
```

The name of the class (with an optional package name) follows the **OpenId** command. There are parentheses following the class name within which contain arguments for the **OpenId** command. The **OpenId** command takes the following arguments:

id	*Required*. The object identifier value used to find the object.
concurrency	*Optional*. The concurrency setting used to open the object. Refer to Concurrency Settings for possible values.
error	*Optional*. A status code, passed by reference, that contains error information if the operation failed.

If the **OpenId** command cannot open the object, it will return a non-object (null string) value.

The **OpenId** command is polymorphic. If there are multiple types of objects within an extent (subclasses of the same persistent object stored together), then the **OpenId** command will return an instance of the object that corresponds to the given ID value.

The **OpenId** command has exactly the same behavior as invoking the **%OpenId** method in ObjectScript. For compatibility, there is also an **Open** with the same behavior as the **%Open** method in ObjectScript.

9.2.3 Object Life Cycle

An object instance is automatically destroyed when there are no longer an variables or object properties referring to it.

For example, in the following code, setting *obj* to an empty string ("") closes the object as there are no other references to it:

Basic example:

```
person = New Sample.Person()
person = ""
```

9.3 Properties, Methods, and Other Members

You can use the properties and methods of an object using dot syntax:

Basic example:

```
objref.Property
objref.Property = value
objref.Method()
```

If the value preceding the dot operator is not a valid object instance, then a runtime error will occur.

9.3.1 Using Properties

To get or set the value of a property, use the dot operator with a variable that refers to an object:

```
person = OpenId Sample.Person(22)
PrintLn person.Name
person.Name = "El Vez"
person = "" ' don't save changes...
```

You can follow object references using cascading dot syntax:

```
person = OpenId Sample.Person(22)
PrintLn person.Home.City
PrintLn person.Home.State
```

From within an instance method, you use the special variable, *Me*, to access other properties of the same object:

Class Member

```
Method PrintOut() [language = basic]
{
     PrintLn Me.Name
}
```

9.3.2 Calling Instance Methods

To invoke an instance method, simply use the dot operator with a variable that refers to an object:

```
person = New Sample.Person()
PrintLn person.NinetyNine()
```

From within an instance method, you use the special variable, *Me*, to invoke other instance methods of the same object:

Class Member

```
Method Test() [language = basic]
{
     Me.Test2()
}
```

9.3.3 Calling Class Methods

To call a class method (a method that can be invoked without having an object instance) use the following syntax:

Basic example:

```
"MyApp.MyClass".MyMethod(22)
```

The name of the class (and optional package name) is placed within "" and followed by the dot operator and the method.

When calling a method, arguments are optional; the parentheses are required. If you specify arguments they must match in number and in sequence the arguments in the class method definition.

To omit an argument value, you must specify an undefined variable. This is a significant difference between ObjectScript and Caché Basic. In ObjectScript a method with an omitted argument can be specified using a placeholder comma, as shown in the following example:

ObjectScript

```
; ObjectScript example
SET tStatement = ##class(%SQL.Statement).%New(,"Sample")
WRITE !,"Success"
```

In Caché Basic, you cannot use a placeholder comma; you must supply an undefined variable, as shown in the following example:

```
' Cache Basic example
ERASE dummy
tStatement = New %SQL.Statement(dummy,"Sample")
PrintLn "Success"
```

In both languages, you do not have to specify trailing arguments. An undefined or omitted argument take the default value for that argument.

9.3.4 The With Statement

The With statement allows you to perform a series of statements on a specified object instance without requalifying the name of the object variable. For example:

```
person = OpenId Sample.Person(22)
With person
    PrintLn .Name
    PrintLn .SSN
    PrintLn .Home.City
End With
```

9.3.5 Me.%GetParameter("Parameter") Syntax

The `obj.%GetParameter` syntax allows for references to class parameters from within the methods of a class. For example, if a class definition include the following parameter and method:

Class Member

```
Parameter MyParam = 22;
```

and the following method:

Class Member

```
ClassMethod WriteMyParam() [ Language = basic ]
{
 Print Me.%GetParameter("MyParam")
}
```

Then the **WriteMyParam** method invokes the **Print** command with the value of the *MyParam* parameter as its argument.

10

Error Handling

This chapter describes the mechanisms Caché Basic provides for handling application errors.

10.1 On Error Goto

The **On Error Goto** statement lets you define an action to take should a runtime error occur:

```
On Error Goto MyError
x = 1 / 0 'induce an error

PrintLn "We will never get here..."
' ...

MyError:
    PrintLn "Error: " & Err.Description
```

When a runtime error occurs, execution will jump to the local label specified by the **On Error Goto** statement. The *Err* object (see below) will contain information about the error.

10.2 The Err Object

The *Err* object is a built-in object that contains information about the current error. Typically you use this within an **On Error** handler.

For more information refer to the Err Object reference page.

10.3 The System.Status Object

Many of the methods in the Caché class library return success or failure information via the %Status data type. For example, the **%Save** method, used to save an instance of %Persistent object to the database, returns a %Status value indicating whether or not the object was saved.

With Caché Basic you can use the *System.Status* object to inspect and manipulate %Status values.

For example, the following code tries to save an object that has missing required values:

```
person = New Sample.Person()
person.Name = "Nobody"
person.SSN  = "" ' required!

' Save this object
status = person.%Save()

If (System.Status.IsError(status)) Then
    System.Status.DisplayError(status)
End If
```

For more information, refer to the System Object reference page as well as the %SYSTEM.Status class.

11

Frequently Asked Questions About Caché Basic

General

Does Caché Basic work only on Windows platforms?

No, Caché Basic is completely platform-independent and works on any platform on which Caché runs. Caché Basic is built into the Caché kernel in the same way as ObjectScript and is not dependent on any external script engine, such as the Microsoft Scripting Host.

Is Caché Basic slower than ObjectScript?

No, Caché Basic source is compiled into the same object code as ObjectScript, so the performance of Caché Basic is the same (with minor variations) as that of ObjectScript.

On what breed of Basic is Caché Basic based?

The syntax of Caché Basic is based on Microsoft VBScript. However, since Caché Basic is a server-side scripting language, tightly integrated with the database, there are a number of differences between these two implementations. For instance, there are no **MsgBox** or **InputBox** functions in Caché Basic, but there are a number of enhancements; for example, the ability to work directly with Caché-specific data types, such as globals, lists, and so on.

Does the introduction of Caché Basic mean that InterSystems is planning to drop support of ObjectScript?

No, the main goal of Caché Basic is to ease the Caché learning curve for developers already familiar with Basic implementations such as Microsoft Visual Basic, not to replace ObjectScript. Both ObjectScript and Caché Basic will be supported and coexist indefinitely.

Are there any other goals beyond “easing the learning curve?”

Yes, with the introduction of Caché Basic, companies using Caché will gain a number of benefits, including:

- Ease in hiring and educating new developers.

- Ease in selling applications developed with Caché. End users do not need to study a new language to use and enhance your applications.
- Re-hosting business logic of VB and ASP applications in Caché. You can migrate the business logic of existing applications from the VB client side or middle tier into Caché, achieving greater performance and scalability. The combination of CSP and Caché Basic is a good candidate for migrating Active Server Pages (ASP) applications.
- Access to interesting resources. By browsing source code archives for VB, Caché developers can find a multitude of valuable and entertaining code samples (for example, games).

Programming

Is there a "Caché Basic shell"?

No, but nothing prevents you from implementing your own. Contact an InterSystems representative for an example of a Caché Basic shell.

How do I work with globals in Caché Basic?

You can directly reference globals using the following syntax:

Table 11–1: Caché Basic Syntax Elements

Syntax element	Description
^Global(index)="Value" Println ^Global(index)	direct access to globals
Exists(^a(1))	if value defined
Traverse(^a(""))	move to the next or previous subscript
EraseValue, EraseArray, Merge, Lock, Unlock, Increment()	miscellaneous functions

How do I work with Caché-specific structures such as List and Piece?

There are a number of enhancements in Caché Basic for **List** and **Piece** support. For example:

Basic example:

```
l = ListBuild("blue","red")
Println List(l,1)

p = "blue^red"
noOfItems = Len(p,"^")
Println Piece(p,"^",1)
```

Is there any analog to $Order in Caché Basic?

The **Traverse()** function provides the same functionality as **$Order** in ObjectScript.

Basic example:

```
i = Traverse(^MyData(""))
While (i <> "")
    Println ^MyData(i)
    i = Traverse(^MyData(i))
Wend
```

How do I work with objects in Caché Basic?

Table 11–2: Caché Basic and Objects Quick Reference

Caché Basic syntax	Objects reference
"Basic.Human".ClassMethod()	call class method
obj.Report()	instance method/property
Me.Name="Anton"	current object property/method
obj=New Basic.Human()	create new object
obj=OpenId Basic.Human(1)	open object instance

What is the analog to $this.Method or ..Method in Caché Basic?

Me.Method

How do I work with SQL in Caché Basic?

Use a dynamic query object:

Basic example:

```
result = New %ResultSet()
result.Prepare("SELECT Name,Age FROM Basic.Human WHERE Age<?")
result.Execute(Arg1)

While (result.Next())
    Println result.Data("Name") & ", " & result.Data("Age")
Wend
```

How do I work with files in Caché Basic?

Use %File object:

Basic example:

```
    file = New %File("c:\test.txt")
    file.Open("WSN")
    file.WriteLine("This is a test")
    file.Close()
```

How do I trap errors in Caché Basic?

Use the **On Error Goto** statement:

Basic example:

```
Function ErrorTest(Arg1)
  On Error Goto errorhandler
  return 1/Arg1

errorhandler:
  PrintLn "Error ", Err.Number, " ", Err.Description, " ", Err.Source
  Err.Clear
  return 0

End Function
```

Why don’t I get an <UNDEFINED> error in Caché Basic?

In Basic, each variable is the empty string by default, so instead of an <UNDEFINED> error you get "" when referring to an undefined variable or function.

Use the **Option Explicit** statement to avoid inadvertently referencing undefined variables and functions.

Can I call ObjectScript programs from Caché Basic and vice versa?

Yes, you can call both methods and functions/procedures, regardless of the language they are written in.

In ObjectScript:

ObjectScript

```
do Procedure^BasicRoutine()
do ##class(MyClass).BasicClassMethod()
```

In Caché Basic:

Basic example:

```
Procedure@ObjectScriptRoutine()
"MyClass".ObjectScriptMethod()
```

Why can’t I see local variables outside of the scope of my procedure?

This is correct behavior; the Basic language defines this functionality. The scope of all variables is limited to the procedure or function where it was defined. Exceptions are variable names that start with the % symbol, such as *%myvar*.

How can I convert date and time values to and from $H format?

Available conversion functions are: **DateConvert()**, **TimeConvert()**, and **DateTimeConvert()**. For example:

Basic example:

```
myDate = "07/15/2002"
println DateConvert(myDate, vbToInternal)   ' returns $H format
myHDate = 59000
println DateConvert(myHDate, vbToExternal)
```

Can I use indirection in Caché Basic?

No.

Support

I suspect there is a bug in Caché Basic. What should I do?

Send a description of the bug to the InterSystems Worldwide Response Center (WRC).

I would like a particular feature in Caché Basic. Can InterSystems implement this?

Send your ideas to the InterSystems Worldwide Response Center (WRC).