Using ObjectScript

Version 2025.1
2025-06-03

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

Using ObjectScript
PDF generated on 2025-06-03
InterSystems IRIS® Version 2025.1

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 Figures

List of Tables

1 Introduction to ObjectScript

ObjectScript is a built-in, fully general programming language in InterSystems IRIS® data platform. ObjectScript source code is compiled into object code that executes within the InterSystems IRIS Virtual Machine. This object code is highly optimized for operations typically found within business applications, including string manipulations and database access. ObjectScript programs are completely portable across all platforms supported by InterSystems IRIS.

You can use ObjectScript in any of the following contexts:

- As the implementation language for methods of InterSystems IRIS classes. (Note that class definitions are not formally part of ObjectScript. Rather, you can use ObjectScript within specific parts of class definitions).
- As the implementation language for stored procedures and triggers within InterSystems SQL.
- To create routines.
- Interactively within the ObjectScript shell.

Important: Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. Use explicit parentheses within an expression to force certain operations to be carried out ahead of others.

1.1 Features

Some of the key features of ObjectScript include:

- Native support for objects including methods, properties, and polymorphism
- Support for concurrency control
- A set of commands for dealing with I/O devices
- Support for multidimensional, sparse arrays: both local and global (persistent)
- Support for efficient, Embedded SQL
- Support for indirection as well as runtime evaluation and execution of commands

1.2 Sample Class with ObjectScript

Class definitions are not part of ObjectScript, but can include ObjectScript in multiple places, and classes are compiled into routines and ultimately into the same runtime code. Within a class, the most common use for ObjectScript is as the implementation of a method.

The following shows an sample class that gives us an opportunity to see some common ObjectScript commands, operators, and functions, and to see how code is organized within a method.

Class Definition

```
Class User.DemoClass
{

/// Generate a random number.
/// This method can be called from outside the class.
ClassMethod Random() [ Language = objectscript ]
{
    set rand=$RANDOM(10)+1        ; rand is an integer in the range 1-10
    write "Your random number: "_rand
    set name=..GetNumberName(rand)
    write !, "Name of this number: "_name
}

/// Input a number.
/// This method can be called from outside the class.
ClassMethod Input() [ Language = objectscript ]
{
    read "Enter a number from 1 to 10: ", input
    set name=..GetNumberName(input)
    write !, "Name of this number: "_name
}

/// Given an number, return the name.
/// This method can be called only from within this class.
ClassMethod GetNumberName(number As %Integer) As %Integer [ Language = objectscript, Private ]
{
    set name=$CASE(number,1:"one",2:"two",3:"three",
        4:"four",5:"five",6:"six",7:"seven",8:"eight",
        9:"nine",10:"ten",:"other")
    quit name
}

/// Write some interesting values.
/// This method can be called from outside the class.
ClassMethod Interesting() [ Language = objectscript ]
{
    write "Today's date: "_$ZDATE($HOROLOG,3)
    write !,"Your installed version: "_$ZVERSION
    write !,"Your username: "_$USERNAME
    write !,"Your security roles: "_$ROLES
}

}
```

Note the following highlights:

- The **Random()** and **Input()** methods invoke the **GetNumberName()** method, which is private to this class and cannot be called from outside the class.
- **WRITE**, **QUIT**, **SET**, and **READ** are ObjectScript commands. The language includes other commands to remove variables, commands to control program flow, commands to control I/O devices, commands to manage transactions (possibly nested), and so on.

  The names of commands are not case-sensitive, although they are shown in running text in all upper case by convention.
- The sample includes two ObjectScript operators. The plus sign (+) performs addition, and the underscore (_) performs string concatenation.

  ObjectScript provides the usual operators and some special operators not seen in other languages.

- **$RANDOM**, **$CASE**, and **$ZDATE** are ObjectScript functions.

  The language provides functions for string operations, conversions of many kinds, formatting operations, mathematical operations, and others.

- **$HOROLOG**, **$ZVERSION**, **$USERNAME**, and **$ROLES** are ObjectScript system variables (called *special variables* in InterSystems IRIS). Most special variables contain values for aspects of the InterSystems IRIS operating environment, the current processing state, and so on.

- ObjectScript supports comment lines, block comments, and comments at the end of statements.

We can execute the methods of this class in the ObjectScript shell, as a demonstration. In these examples, `TESTNAMESPACE>` is the prompt shown in the shell. The text after the prompt on the same line is the entered command. The lines after that show the values that the system writes to the shell in response.

```
TESTNAMESPACE>do ##class(User.DemoClass).Input()
Enter a number from 1 to 10: 7
Name of this number: seven
TESTNAMESPACE>do ##class(User.DemoClass).Interesting()
Today's date: 2021-07-15
Your installed version: IRIS for Windows (x86-64) 2019.3 (Build 310U) Mon Oct 21 2019 13:48:58 EDT
Your username: SuperUser
Your security roles: %All
TESTNAMESPACE>
```

1.3 Sample Routine

The following shows an sample ObjectScript routine named `demoroutine`. It contains procedures that do the exact same thing as the methods shown in the sample class in the previous section.

ObjectScript

```
 ; this is demoroutine
 write "Use one of the following entry points:"
 write !,"random"
 write !,"input"
 write !,"interesting"
 quit

 //this procedure can be called from outside the routine
random() public {
    set rand=$RANDOM(10)+1        ; rand is an integer in the range 1-10
    write "Your random number: "_rand
    set name=$$getnumbername(rand)
    write !, "Name of this number: "_name
 }

 //this procedure can be called from outside the routine
input() public {
    read "Enter a number from 1 to 10: ", input
    set name=$$getnumbername(input)
    write !, "Name of this number: "_name
 }

 //this procedure can be called only from within this routine
getnumbername(number) {
    set name=$CASE(number,1:"one",2:"two",3:"three",
        4:"four",5:"five",6:"six",7:"seven",8:"eight",
        9:"nine",10:"ten",:"other")
    quit name
}

 /* write some interesting values
 this procedure can be called from outside the routine
 */
interesting() public {
    write "Today's date: "_$ZDATE($HOROLOG,3)
    write !,"Your installed version: "_$ZVERSION
    write !,"Your username: "_$USERNAME
    write !,"Your security roles: "_$ROLES
    }
```

Note the following highlights:

- The only identifiers that actually start with a caret (`^`) are the names of globals; these are discussed later in this page. However, in running text and in code comments, it is customary to refer to a routine as if its name started with a caret, because you use the caret when you invoke the routine (as shown later in this page). For example, the routine `demoroutine` is usually called `^demoroutine`.
- The routine name does not have to be included within the routine. However, many programmers include the routine name as a comment at the start of the routine or as the first label in the routine.
- The routine has multiple *labels*: `random`, `input`, `getnumbername`, and `interesting`.

  Labels are used to indicate the starting point for procedures (as in this example) and legacy forms of subroutines. You can also use them as a destination for certain commands.

  Labels are common in routines, but you can also use them within methods.

  Labels are also called *entry points* or *tags*.
- The `random` and `input` subroutines invoke the `getnumbername` subroutine, which is private to the routine.

We can execute parts of this routine in the ObjectScript shell, as a demonstration. First, the following shows a session in which we run the routine itself.

```
TESTNAMESPACE>do ^demoroutine
Use one of the following entry points:
random
input
TESTNAMESPACE>
```

When we run the routine, we just get help information, as you can see. It is not required to write your routines in this way, but it is common. Note that the routine includes a **QUIT** before the first label, to ensure that when a user invokes the routine, processing is halted before that label. This practice is also not required, but is also common.

Next, the following shows how a couple of the subroutines behave:

```
TESTNAMESPACE>do input^demoroutine
Enter a number from 1 to 10: 7
Name of this number: seven
TESTNAMESPACE>do interesting^demoroutine
Today's date: 2018-02-06
Your installed version: IRIS for Windows (x86-64) 2018.1 (Build 513U) Fri Jan 26 2018 18:35:11 EST
Your username: _SYSTEM
Your security roles: %All
TESTNAMESPACE>
```

A method can contain the same statements, labels, and comments as routines do. That is, all the information here about the contents of a routine also applies to the contents of a method.

1.4 Variables

In ObjectScript, there are two primary kinds of variables, as categorized by how they hold data. The name of the variable establishes which kind of variable it is.

- *Local variables*, which hold data in memory.

  Local variables can have public or private scope.

  Example local variable names are `MyVar` and `%MyVar`.

- *Global variables*, which hold data in a database. These are also called *globals*. All interactions with a global affect the database immediately. For example, when you set the value of a global, that change immediately affects what is stored; there is no separate step for storing values. Similarly, when you remove a global, the data is immediately removed from the database.

  Example global variable names are `^MyVar` and `^%MyVar`.

See ObjectScript Variables and Scope.

1.5 Multidimensional Arrays

In ObjectScript, any variable can be an InterSystems IRIS *multidimensional array* (also called an *array*). An object property can also be a multidimensional array, if it is declared as such. A multidimensional array is generally intended to hold a set of values that are related in some way. ObjectScript provides commands and functions that provide convenient and fast access to the values.

You may or may not work directly with multidimensional arrays, depending on the APIs that you use and your own preferences. InterSystems IRIS provides a class-based alternative to use when you want a container for sets of related values; see Collection Classes.

1.5.1 Basics

A multidimensional array consists of any number of *nodes*, defined by subscripts. The following example sets several nodes of an array and then prints the contents of the array:

ObjectScript

```
 set myarray(1)="value A"
 set myarray(2)="value B"
 set myarray(3)="value C"
 zwrite myarray
```

This example shows a typical array. Notes:

- This array has one subscript. In this case, the subscripts are the integers 1, 2, and 3.
- There is no need to declare the structure of the array ahead of time.
- `myarray` is the name of the array itself.
- ObjectScript provides commands and functions that can act on an entire array or on specific nodes. For example:

  ObjectScript

  ```
   kill myarray
  ```

  You can also kill a specific node and its child nodes.
- The following variation sets several subscripts of a global array named `^myglobal`; that is, these values are written to disk:

  ObjectScript

  ```
   set ^myglobal(1)="value A"
   set ^myglobal(2)="value B"
   set ^myglobal(3)="value C"
  ```

- There is a limit to the possible length of a global reference. This limit affects the length of the global name and the length and number of any subscripts. If you exceed the limit, you get a <SUBSCRIPT> error. See Maximum Length of a Global Reference.
- The length of a value of a node must be less than the string length limit.

A multidimensional array has one reserved memory location for each defined node and no more than that. For a global, all the disk space that it uses is dynamically allocated.

1.5.2 Structure Variations

The preceding examples show a common form of array. Note the following possible variations:

- You can have any number of subscripts. For example:

  ObjectScript

  ```
  Set myarray(1,1,1)="grandchild of value A"
  ```

- A subscript can be a string. The following is valid:

  ObjectScript

  ```
  set myarray("notes to self","2 Dec 2010")="hello world"
  ```

1.5.3 Use Notes

For those who are learning ObjectScript, a common mistake is to confuse globals and arrays. It is important to remember that any variable is either local or global, *and* may or may not have subscripts. The following table shows the possibilities:

Kind of Variable	Example and Notes
Local variable without subscripts	`Set MyVar=10` Variables like this are quite common. The majority of the variables you see might be like this.

Local variable with subscripts	`Set MyVar(1)="alpha"`
	`Set MyVar(2)="beta"`
	`Set MyVar(3)="gamma"`
	A local array like this is useful when you want to pass a set of related values.

Global variable without subscripts	`Set ^MyVar="saved note"` In practice, globals usually have subscripts.
Global variable with subscripts	`Set ^MyVar($USERNAME,"Preference 1")=42`

1.6 Operators

This section provides an overview of the operators in ObjectScript; some are familiar, and others are not.

Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. You can use explicit parentheses within an expression to force certain operations to be carried out ahead of others.

Typically you use parentheses even where you do not strictly need them. It is useful to other programmers (and to yourself at a later date) to do this because it makes the intent of your code clearer.

1.6.1 Familiar Operators

ObjectScript provides the following operators for common activities:

- Mathematical operators: addition (+), subtraction (-), division (/), multiplication (*), integer division (\), modulus (#), and exponentiation (**)
- Unary operators: positive (+) and negative (-)
- String concatenation operator (_)
- Logical comparison operators: equals (=), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=)
- Logical complement operator (')

  You can use this immediately before any logical value as well as immediately before a logical comparison operator.
- Operators to combine logical values: AND (&&), OR (||)

  Note that ObjectScript also supports an older, less efficient form of each of these: & is a form of the && operator, and ! is a form of the || operator. You might see these older forms in existing code.

1.6.2 Unfamiliar Operators

ObjectScript also includes operators that have no equivalent in some languages. The most important ones are as follows:

- The pattern match operator (?) tests whether the characters in its left operand use the pattern in its right operand. You can specify the number of times the pattern is to occur, specify alternative patterns, specify pattern nesting, and so on.

  For example, the following writes the value 1 (true) if a string (`testthis`) is formatted as a U.S. Social Security Number and otherwise writes 0.

  ObjectScript

  ```
   Set testthis="333-99-0000"
   Write testthis ?3N1"-"2N1"-"4N
  ```

  This is a valuable tool for ensuring the validity of input data, and you can use it within the definition of class properties.
- The binary contains operator ([) returns 1 (true) or 0 (false) depending on whether the sequence of characters in the right operand is a substring of the left operand. For example:

  ObjectScript

  ```
   Set L="Steam Locomotive",S="Steam"
   Write L[S
  ```
- The binary follows operator (]) tests whether the characters in the left operand come after the characters in the right operand in ASCII collating sequence.
- The binary sorts after operator (]]) tests whether the left operand sorts after the right operand in numeric subscript collation sequence.

- The indirection operator (`@`) allows you to perform dynamic runtime substitution of part or all of a command argument, a variable name, a subscript list, or a pattern. InterSystems IRIS performs the substitution before execution of the associated command.

1.7 Commands

This section provides an overview of the commands that you are most likely to use and to see in ObjectScript. These include commands that are similar to those in other languages, as well as others that have no equivalent in other languages.

The names of commands are not case-sensitive, although they are shown in running text in all upper case by convention.

1.7.1 Familiar Commands

ObjectScript provides commands to perform familiar tasks such as the following:

- To define variables, use **SET** as shown previously.
- To remove variables, use **KILL** as shown previously.
- To control the flow of logic, use the following commands:
  - **IF**, **ELSEIF**, and **ELSE**, which work together
  - **FOR**
  - **WHILE**, which can be used on its own
  - **DO** and **WHILE**, which can be used together
  - **QUIT**, which can also return a value

  There are other commands for flow control, but they are used less often.
- To trap errors, use **TRY** and **CATCH**, which work together. See Using TRY-CATCH.
- To write a value, use **WRITE**. This writes values to the current device (for example, the Terminal or a file).

  Used without an argument, this command writes the values of all local variables. This is particularly convenient in the Terminal.

  This command can use a small set of format control code characters that position the output. In existing code, you are likely to see the exclamation point, which starts a new line. For example:

  ObjectScript

  ```
  write "hello world",!,"another line"
  ```
- To read a value from the current device (for example, the Terminal), use **READ**.
- To work with devices other than the principal device, use the following commands:
  - **OPEN** makes a device available for use.
  - **USE** specifies an open device as the current device for use by **WRITE** and **READ**.
  - **CLOSE** makes a device no longer available for use.
- To control concurrency, use **LOCK**. Note that the InterSystems IRIS lock management system is different from analogous systems in other languages. It is important to review how it works; see Locking and Concurrency Control.

- To manage transactions, use **TSTART**, **TCOMMIT**, **TROLLBACK**, and related commands. See Transaction Processing.
- For debugging, use **ZBREAK** and related commands.
- To suspend execution, use **HANG**.

1.7.2 Commands for Use with Multidimensional Arrays

In ObjectScript, you can work with multidimensional arrays in the following ways:

- To define nodes, use the **SET** command.
- To remove individual nodes or all nodes, use the **KILL** command.

  For example, the following removes an entire multidimensional array:

  ObjectScript

  ```
   kill myarray
  ```

  In contrast, the following removes the node `myarray("2 Dec 2010")` and all its children:

  ObjectScript

  ```
   kill myarray("2 Dec 2010")
  ```

- To delete a global or a global node but none of its descendent subnodes, use **ZKILL**.
- To iterate through all nodes of a multidimensional array and write them all, use **ZWRITE**. This is particularly convenient in the Terminal. The following sample Terminal session shows what the output looks like:

  ```
  TESTNAMESPACE>ZWRITE ^myarray
  ^myarray(1)="value A"
  ^myarray(2)="value B"
  ^myarray(3)="value C"
  ```

  This example uses a global variable rather than a local one, but remember that both can be multidimensional arrays.

- To copy a set of nodes from one multidimensional array into another, preserving existing nodes in the target if possible, use **MERGE**. For example, the following command copies an entire in-memory array (`sourcearray`) into a new global (`^mytestglobal`):

  ObjectScript

  ```
   MERGE ^mytestglobal=sourcearray
  ```

  This can be a useful way of examining the contents of an array that you are using, while debugging your code.

1.8 System Functions

This section introduces some of the most commonly used ObjectScript functions, grouped by purpose. The names of these functions are not case-sensitive.

- Choosing values: **$SELECT** and **$CASE**
- Testing for existence of a variable (or of a node of a variable): **$DATA** and **$GET**

- Creating and working with native-list format lists: **$LISTBUILD**, **$LISTGET**, **$LIST**, and others. See Lists in ObjectScript, which also introduces alternatives.
- Working with multidimensional arrays: **$ORDER**, **$QUERY**, **$DATA**, and **$GET**. See Multidimensional Arrays, which also introduces alternatives.
- Creating characters that cannot be typed, for inclusion in strings: **$CHAR**. Given an integer, **$CHAR** returns the corresponding ASCII or Unicode character. Common uses:
  - **$CHAR(9)** is a tab.
  - **$CHAR(10)** is a line feed.
  - **$CHAR(13)** is a carriage return.
  - **$CHAR(13,10)** is a carriage return and line feed pair.

  The function **$ASCII** returns the ASCII value of the given character. See Strings in ObjectScript.

For a compact, full list, see ObjectScript Function Reference.

The InterSystems IRIS class library also provides a large set of APIs. See the InterSystems Programming Tools Index.

1.9 Special Variables

This section introduces some InterSystems IRIS *special variables*. The names of these variables are not case-sensitive.

Some special variables provide information about the environment in which the code is running. These include the following:

- **$HOROLOG**, which contains the date and time for the current process, as given by the operating system. See Date and Time Values.
- **$USERNAME** and **$ROLES**, which contain information about the username currently in use, as well as the roles to which that user belongs.

  ObjectScript

  ```
   write "You are logged in as: ", $USERNAME, !, "And you belong to these roles: ",$ROLES
  ```
- **$ZVERSION**, which contains a string that identifies the currently running version of InterSystems IRIS.

Others include **$JOB**, **$ZTIMEZONE**, **$IO**, and **$ZDEVICE**.

Other variables provide information about the processing state of the code. These include **$STACK**, **$TLEVEL**, **$NAMESPACE**, and **$ZERROR**.

1.9.1 $SYSTEM Special Variable

The special variable **$SYSTEM** provides easy access to a large set of utility methods.

The special variable **$SYSTEM** is an alias for the %SYSTEM package, which contains classes that provide class methods that address a wide variety of needs. The customary way to refer to methods in %SYSTEM is to build a reference that uses the **$SYSTEM** variable. For example, the following command executes the **SetFlags()** method in the %SYSTEM.OBJ class:

ObjectScript

```
 DO $SYSTEM.OBJ.SetFlags("ck")
```

Because names of special variables are not case-sensitive (unlike names of classes and their members), the following commands are all equivalent:

ObjectScript

```
 DO ##class(%SYSTEM.OBJ).SetFlags("ck")
 DO $System.OBJ.SetFlags("ck")
 DO $SYSTEM.OBJ.SetFlags("ck")
 DO $system.OBJ.SetFlags("ck")
```

The classes all provide the **Help()** method, which can print a list of available methods in the class. For example:

```
TESTNAMESPACE>do $system.OBJ.Help()
'Do $system.OBJ.Help(method)' will display a full description of an individual method.

Methods of the class: %SYSTEM.OBJ

CloseObjects()
     Deprecated function, to close objects let them go out of scope.

Compile(classes,qspec,&errorlog,recurse)
     Compile a class.

CompileAll(qspec,&errorlog)
     Compile all classes within this namespace
....
```

You can also use the name of a method as an argument to **Help()**. For example:

```
TESTNAMESPACE>d $system.OBJ.Help("Compile")
Description of the method:class Compile:%SYSTEM.OBJ

Compile(classes:%String="",qspec:%String="",&errorlog:%String,recurse:%Boolean=0)
Compile a class.
<p>Compiles the class <var>classes</var>, which can be a single class, a comma separated list,
a subscripted array of class names, or include wild cards. If <var>recurse</var> is true then
do not output the intial 'compiling' message or the compile report as this is being called inside
another compile loop.<br>
<var>qspec</var> is a list of flags or qualifiers which can be displayed with
'Do $system.OBJ.ShowQualifiers()'
and 'Do $system.OBJ.ShowFlags()
```

1.10 Potential Pitfalls

The following items can confuse programmers who are new to ObjectScript, particularly if those who are responsible for maintaining code written by other programmers:

- Within a routine or a method, every line must be indented by at least one space or one tab unless that line contains a label. That is, if there is text of any kind in the first character position, the compiler and your IDE treat it as a label.

  There is one exception: A curly brace is accepted in the first character position.

- There must be exactly one space (not a tab) between a command and its first argument. Otherwise, your IDE indicates that you have a syntax error.

  Similarly, the ObjectScript shell displays a syntax error as follows:

  ```
  TESTNAMESPACE>write  5

  WRITE  5
         ^
  <SYNTAX>
  TESTNAMESPACE>
  ```

- Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. You can use explicit parentheses within an expression to force certain operations to be carried ahead of others.

Typically you use parentheses even where you do not strictly need them. It is useful to other programmers (and to yourself at a later date) to do this because it makes the intent of your code clearer.

- For reasons of history, ObjectScript does not consider an empty string (`""`) to equal the ASCII NULL value. To represent the ASCII NULL value, use `$CHAR(0)`. (**$CHAR** is a system function that returns an ASCII character, given its decimal-based code.) For example:

  ObjectScript

  ```
  write "" = $char(0)
  ```

  Similarly, when ObjectScript values are projected to SQL or XML, the values `""` and `$CHAR(0)` are treated differently. For information on the SQL projections of these values, see Null and the Empty String. For information on the XML projections of these values, see Handling Empty Strings and Null Values.

- Some parts of ObjectScript are case-sensitive while others are not. The case-insensitive items include names of commands, functions, special variables, namespaces, and users.

  The case-sensitive items include names of most of the elements that you define: routines, variables, classes, properties, and methods. For more details, see Syntax Rules.

- Most command names can be represented by an abbreviated form. Therefore, `WRITE`, `Write`, `write`, `W`, and `w` are all valid forms of the **WRITE** command. For a list, see Abbreviations Used in ObjectScript.

- For many of the commands, you can include a *postconditional expression* (often simply called a *postconditional*).

  This expression controls whether InterSystems IRIS executes the command. If the postconditional expression evaluates to true (nonzero), InterSystems IRIS executes the command. If the expression evaluates to false (zero or null), InterSystems IRIS ignores the command and continues with the next command.

  For example:

  ObjectScript

  ```
  Set count = 6
  Write:count<5 "Print this if count is less than five"
  Write:count>5 "Print this if count is greater than five"
  ```

  The preceding generates the following output: `Print this if count is greater than five`

  Note: If postconditionals are new to you, you might find the phrase "postconditional expression" somewhat misleading, because it suggests (incorrectly) that the expression is executed *after* the command. Despite the name, a postconditional is executed *before* the command.

  Postconditionals are also permitted for specific arguments of specific commands.

- You can include multiple commands on a single line. For example:

  ObjectScript

  ```
  set myval="hello world" write myval
  ```

  When you do this, beware that you must use two spaces after any command that does not take arguments, if there are additional commands on that line; if you do not do so, a syntax error occurs.

- The **IF**, **ELSE**, **FOR**, and **DO** commands are available in two forms:

  – A newer block form, which uses curly braces to indicate the block. For example:

ObjectScript

```
if (testvalue=1) {
    write "hello world"
 }
```

InterSystems recommends that you use the block form in all new code.

– An older line-based form, which does not use curly braces. For example:

ObjectScript

```
if (testvalue=1) write "hello world"
```

- As a result of the preceding items, ObjectScript can be written in a very compact form. For example:

  ObjectScript

  ```
  s:$g(%d(3))'="" %d(3)=$$fdN3(%d(3)) q
  ```

  The class compiler automatically generates compact code of the form shown above (although not necessarily with abbreviated commands as in this example). Sometimes it is useful to look at this generated code, to track down the source of a problem or to understand how something works.

- There are no truly reserved words in ObjectScript, so it is theoretically possible to have a variable named `set`, for example. However, it is prudent to avoid names of commands, functions, SQL reserved words, and certain system items; see Syntax Rules.

- InterSystems IRIS allocates a fixed amount of memory to hold the results of string operations. If a string expression exceeds the amount of space allocated, a <MAXSTRING> error results. See string length limit.

  For class definitions, the string operation limit affects the size of string properties. InterSystems IRIS provides a system object (called a *stream*) that you can use when you need to work with strings that exceed this limit; in such cases, you use the stream interface classes.

1.11 See Also

To learn more about ObjectScript, you can also refer to:

- The ObjectScript Tutorial for an interactive introduction to most language elements.
- The ObjectScript Reference for details on individual commands and functions.

2 ObjectScript Syntax Basics

This topic describes the basic rules of ObjectScript syntax. Additional pages describe more syntax rules.

2.1 Left-to-Right Precedence

Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. This is different from other languages in which certain operators have higher precedence than others. For more information, see Operator Precedence.

2.2 Case Sensitivity

Some parts of ObjectScript are case-sensitive while others are not. The following overview covers many but not all the cases:

- *Not case-sensitive*: ObjectScript commands, functions, and system variables are not case sensitive.
- *Usually not case-sensitive*: Case sensitivity of the following is platform-dependent: device names, file names, directory names, disk drive names. The exponent symbol is usually not case-sensitive, but this depends on your system configuration.
- *Case-sensitive*: variable names (other than the ObjectScript system variables) are case-sensitive. Names of routines and their entry points, names of include files and macros are also all case sensitive.

Also, when you use ObjectScript syntax to refer to a package, class, or class member, you must match its case exactly. However, you cannot create packages or classes that differ only in case. For example, if you create the class `A.B`, you must use that case when referring to that class, for example: `##class(A.B).MyMethod()`. However, you cannot now create classes named `a.b`, `A.b`, or `a.B`; you also cannot create new classes in any packages named `a.b`, `A.b`, or `a.B`.

2.3 Identifiers

An *identifier* is the name of a variable, package, class, class member, routine, or label. In general, legal identifiers consist of letter and number characters; with few exceptions, punctuation characters are not permitted in identifiers. Identifiers are case-sensitive.

Note: SQL identifiers, in contrast, are *not* case-sensitive.

For variables, the variable name determines the kind of variable it is, which determines its scope and special characteristics. See Variables.

For reference information on identifiers, including variable names to avoid, see Rules and Guidelines for Identifiers.

2.4 Reserved Words

There are no reserved words in ObjectScript; you can use any valid identifier as a variable name, function name, or label. At the same time, it is best to avoid using identifiers that are command names, function names, or other such strings. Also, since ObjectScript code includes support for embedded SQL, it is prudent to avoid naming any function, object, variable, or other entity with an SQL reserved word, as this may cause difficulties elsewhere.

2.5 Expressions

An ObjectScript expression is one or more *tokens* that can be evaluated to yield a value. The simplest expression is simply a literal or variable:

ObjectScript

```
 SET expr = 22
 SET expr = "hello"
 SET expr = x
```

You can create more complex expressions using operators and functions:

ObjectScript

```
 SET expr = +x
 SET expr = x + 22
 SET expr = array(1)
 SET expr = ^data("x",1)
 SET expr = $Length(x)
```

An expression may consist of, or include, an object property, instance method call, or class method call:

ObjectScript

```
 SET expr = person.Name
 SET expr = obj.Add(1,2)
 SET expr = ##class(MyApp.MyClass).Method()
```

You can directly invoke an ObjectScript routine call within an expression by placing $$ in front of the routine call:

ObjectScript

```
 SET expr = $$MyFunc^MyRoutine(1)
```

2.6 Commands

All the execution tasks in ObjectScript are performed by *commands*. Every command consists of a command keyword followed by (in most cases) one or more command arguments. Note the following syntax rules:

- Command names are not case-sensitive. Most command names can be represented by an abbreviated form. Therefore, WRITE, Write, write, W, and w are all valid forms of the **WRITE** command. For a list of command abbreviations, see Table of Abbreviations.
- Command keywords are not reserved words. It is therefore possible to use a command keyword as a user-assigned name for a variable, label, or other identifier.
- Within code, an ObjectScript command cannot appear in column 1; see the discussion later on whitespace.

  The same restriction does not apply when issuing a command from the ObjectScript shell or the **XECUTE** command.
- If the command accepts arguments, there must be exactly 1 space between the command and the first argument.
- You can include multiple commands (with their arguments) on the same line.

  The commands are executed in strict left-to-right order, and are functionally identical to commands appearing on separate lines. A command with arguments must be separated from the command following it by one space character. An argumentless command must be separated from the command following it by two space characters. A label can be followed by one or more commands on the same line. A comment can follow one or more commands on the same line.

  For the maximum length of a line of source code, see General System Limits.

  One or more commands may follow a label on the same line; the label and the command are separated by one or more spaces.
- No end-of-command or end-of-line delimiter is required or permitted. You can specify an in-line comment following a command, indicating that the rest of the command line is a comment. A blank space is required between the end of a command and comment syntax, with the exception of `##;` and `/* comment */` syntax. A `/* comment */` multiline comment can be specified within a command as well as at the end of one.
- Many commands can use a postconditional expression; this phrase refers to a logical expression that determines whether to execute the command. If the expression evaluates to true, the command is executed; otherwise it isn’t.

2.7 Command Postconditional Expressions

In most cases, when you specify an ObjectScript command you can append a *postconditional*.

A postconditional is an optional expression that is appended to a command or (in some cases) a command argument that controls whether InterSystems IRIS executes that command or command argument. If the postconditional expression evaluates to TRUE (defined as nonzero), InterSystems IRIS executes the command or the command argument. If the postconditional expression evaluates to FALSE (defined as zero), InterSystems IRIS does not execute the command or command argument, and execution continues with the next command or command argument.

All ObjectScript commands can take a postconditional expression, except the flow-of-control commands (**IF**, **ELSEIF**, and **ELSE**; **FOR**, **WHILE**, and **DO WHILE**) and the block structure error handling commands (**TRY**, **CATCH**).

The ObjectScript commands **DO** and **XECUTE** can append postconditional expressions both to the command keyword and to their command arguments. A postconditional expression is always optional; for example, some of the command’s arguments may have an appended postconditional while its other arguments do not.

If both a command keyword and one or more of that command's arguments specify a postconditionals, the keyword postconditional is evaluated first. Only if this keyword postconditional evaluates to TRUE are the command argument postconditionals evaluated. If a command keyword postconditional evaluates to FALSE, the command is not executed and program execution continues with the next command. If a command argument postconditional evaluates to FALSE, the argument is not executed and execution of the command continues with the next argument in left-to-right sequence.

2.7.1 Postconditional Syntax

To add a postconditional to a command, place a colon (:) and an expression immediately after the command keyword, so that the syntax for a command with a postconditional expression is:

```
Command:pc
```

where *Command* is the command keyword, the colon is a required literal character, and *pc* can be any valid expression.

A command postconditional must follow these syntax rules:

- No spaces, tabs, line breaks, or comments are permitted between a command keyword and its postconditional, or between a command argument and its postconditional. No spaces are permitted before or after the colon character.
- No spaces, tabs, line breaks, or comments are permitted within a postconditional expression, unless either an entire postconditional expression is enclosed in parentheses or the postconditional expression has an argument list enclosed in parentheses. Spaces, tabs, line breaks, and comments are permitted within parentheses.
- Spacing requirements following a postconditional expression are the same as those following a command keyword: there must be exactly one space between the last character of the keyword postconditional expression and the first character of the first argument; for argumentless commands, there must be two or more spaces between the last character of the postconditional expression and the next command on the same line, unless the postconditional is immediately followed by a close curly brace. (If parentheses are used, the closing parenthesis is treated as the last character of the postconditional expression.)

Note that a postconditional expression is not technically a command argument (though in the ObjectScript reference pages the explanation of the postconditional is presented as part of the Arguments section). A postconditional is always optional.

2.7.2 Evaluation of Postconditionals

InterSystems IRIS evaluates a postconditional expression as either True or False. Most commonly these are represented by 1 and 0, which are the recommended values. However, InterSystems IRIS performs postconditional evaluation on any value, evaluating it as False if it evaluates to 0 (zero), and True if it evaluates to a nonzero value.

- InterSystems IRIS evaluates as True any valid nonzero numeric value. It uses the same criteria for valid numeric values as the arithmetic operators. Thus, the following all evaluate to True: 1, “1”, 007, 3.5, -.007, 7.0 , 3 little pigs, $CHAR(49), 0_"1".
- InterSystems IRIS evaluates as False the value zero (0), and any nonnumeric value, including a null string ("") or a string containing a blank space (" "). Thus, the following all evaluate to False: 0, -0.0, A, -, $, The 3 little pigs, $CHAR(0), $CHAR(48), "0_1".
- Standard equivalence rules apply. Thus, the following evaluate to True: 0=0, 0="0", "a"=$CHAR(97), 0=$CHAR(48), and (" "=$CHAR(32)). The following evaluate to False: 0="", 0=$CHAR(0), and (""=$CHAR(32)).

In the following example, which **WRITE** command is executed depends on the value of the variable *count*:

ObjectScript

```
FOR count=1:1:10 {
  WRITE:count<5 count," is less than 5",!
  WRITE:count=5 count," is 5",!
  WRITE:count>5 count," is greater than 5",!
}
```

2.8 Command Arguments

Following a command keyword, there can be zero, one, or multiple arguments that specify the object(s) or the scope of the command. If a command takes one or more arguments, you must include exactly one space between the command keyword and the first argument. For example:

ObjectScript

```
SET x = 2
```

Spaces can appear within arguments or between arguments, so long as the first character of the first argument is separated from the command itself by exactly one space (as appears above). Thus the following are all valid:

ObjectScript

```
 SET a = 1
 SET b=2
 SET c=3,d=4
 SET e= 5   ,  f =6
 SET g
     =            7
 WRITE a,b,c,d,e,f,g
```

If a command takes a postconditional expression, there must be no spaces between the command keyword and the postconditional, and there must be exactly one space between the postconditional and the beginning of the first argument. Thus, the following are all valid forms of the **QUIT** command:

ObjectScript

```
QUIT x+y
QUIT x + y
QUIT:x<0
QUIT:x<0 x+y
QUIT:x<0 x + y
```

No spaces are required between arguments, but multiple blank spaces can be used between arguments. These blank spaces have no effect on the execution of the command. Line breaks, tabs, and comments can also be included within or between command arguments with no effect on the execution of the command. For further details, see White Space.

2.8.1 Multiple Arguments

Many commands allow you to specify multiple independent arguments. The delimiter for command arguments is the comma ,. That is, you specify multiple arguments to a single command as a comma-separated list following the command. For example:

ObjectScript

```
  SET x=2,y=4,z=6
```

This command uses three arguments to assign values to the three specified variables. In this case, these multiple arguments are repetitive; that is, the command is applied independently to each argument in the order specified. Internally, InterSystems

IRIS® data platform parses this as three separate **SET** commands. When debugging, each of these multiple arguments is a separate step.

In the command syntax provided in the command reference pages, arguments that can be repeated are followed by a comma and ellipsis: `,...`. The comma is a required delimiter character for the argument, and the ellipsis (...) indicates that an unspecified number of repetitive arguments can be specified.

Repetitive arguments are executed in strict left-to-right order. Therefore, the following command is valid:

ObjectScript

```
 SET x=2,y=x+1,z=y+x
```

but the following command is not valid:

ObjectScript

```
 SET y=x+1,x=2,z=y+x
```

Because execution is performed independently on each repetitive argument, in the order specified, valid arguments are executed until the first invalid argument is encountered. In the following example, `SET x` assigns a value to *x*, `SET y` generates an <UNDEFINED> error, and because `SET z` is not evaluated, the <DIVIDE> (divide-by-zero) error is not detected:

ObjectScript

```
 KILL x,y,z
 SET x=2,y=z,z=5/0
 WRITE "x is:",x
```

2.8.2 Arguments with Their Own Arguments and Postconditionals

Some command arguments accept their own *arguments* and even their own postconditionals.

The following sample command shows a comma between the two arguments. Each argument is followed by a colon-separated list of its own arguments:

ObjectScript

```
VIEW X:y:z:a,B:a:y:z
```

For a few commands (**DO**, **XECUTE**, and **GOTO**), a colon following an argument specifies a postconditional expression that determines whether or not that argument should be executed.

2.8.3 Argumentless Commands

Commands that do not take an argument are referred to as *argumentless* commands. A postconditional expression appended to the keyword is not considered an argument.

There are a small number of commands that are always argumentless. For example, **HALT**, **CONTINUE**, **TRY**, **TSTART**, and **TCOMMIT** are argumentless commands.

Several commands are optionally argumentless. For example, **BREAK**, **CATCH**, **FOR**, **GOTO**, **KILL**, **LOCK**, **NEW**, **QUIT**, **RETURN**, **TROLLBACK**, **WRITE**, and **ZWRITE** all have argumentless syntactic forms. In such cases, the argumentless command may have a slightly different meaning than the same command with an argument.

If you use an argumentless command at the end of the line, trailing spaces are not required. If you use an argumentless command on the same code line as other commands, you must place *two* (or more) spaces between the argumentless command and any command that follows it. For example:

ObjectScript

```
QUIT:x=10  WRITE "not 10 yet"
```

In this case, **QUIT** is an argumentless command with a postconditional expression, and a minimum of two spaces is required between it and the next command.

2.8.3.1 Argumentless Commands and Curly Braces

Argumentless commands when used with command blocks delimited by curly braces do not have whitespace restrictions:

- An argumentless command that is immediately followed by an opening curly brace has no whitespace requirement between the command name and the curly brace. You can specify none, one, or more than one spaces, tabs, or line returns. This is true both for argumentless commands that can take an argument, such as **FOR**, and argumentless commands that cannot take an argument, such as **ELSE**.

  ObjectScript

  ```
  FOR  {
     WRITE !,"Quit out of 1st endless loop"
     QUIT
  }
  FOR{
     WRITE !,"Quit out of 2nd endless loop"
     QUIT
  }
  FOR
  {
     WRITE !,"Quit out of 3rd endless loop"
     QUIT
  }
  ```

- An argumentless command that is immediately followed by a closing curly brace does not require trailing spaces, because the closing curly brace acts as a delimiter. For example, the following is a valid use of the argumentless **QUIT**:

  ObjectScript

  ```
  IF 1=2 {
     WRITE "Math error"}
  ELSE {
     WRITE "Arthmetic OK"
     QUIT}
  WRITE !,"Done"
  ```

2.9 Routine Syntax

Routines are the building blocks of ObjectScript programs. Class definitions (not formally part of the ObjectScript language) are compiled into routines. You can write any combination of routines and class definitions that meets your needs; all this code is ultimately compiled into runtime code.

A routine is a unit of code (generally seen a single unit within an IDE or a source control system) that consists of lines of ObjectScript code, comments, and blank lines. It is generally organized as follows:

1. One or more comment lines at the start, optionally indicating the name of the routine as well as containing any usage notes or update history.
2. Any number of procedures, which have the following syntax:

   ```
   ProcedureName(Arguments) scopekeyword {
       //procedure implementation
   }
   ```

Where:

- *ProcedureName* is the name of the procedure. This is an ObjectScript label and it must start in column 1 of the line.
- *Arguments* is an optional comma-separated list of arguments. Even if there are no arguments, you must include the parentheses.
- The optional *scopekeyword* is one of the following (not case-sensitive):
  - `Public`. If you specify `Public`, then the procedure is *public* and can be invoked outside of the routine itself.
  - `Private` (the default for procedures). If you specify `Private`, the procedure is *private* and can be invoked only by other code in the same routine.
- The implementation consists of zero or more lines of ObjectScript. This code can return a value via RETURN or QUIT.

For details, see Defining Procedures.

The routine can also contain:

- Comments and whitespace, both within procedures and between them
- Legacy forms of subroutines

2.10 Method Syntax

For completeness, it is also useful to consider methods, which are defined within classes. Class definitions are not part of ObjectScript, but can include ObjectScript in multiple places, and classes are compiled into routines and ultimately into the same runtime code. The most common use for ObjectScript in a class is within methods. A class method (called a static method in some languages) has the following syntax:

```
ClassMethod MethodName(Arguments) as Classname [ Keywords]
{
 //method implementation
}
```

The syntax for an instance method (relevant only in object classes) is similar:

```
Method MethodName(Arguments) as Classname [ Keywords]
{
 //method implementation
}
```

Where:

- *MethodName* is the name of the method.
- *Arguments* is a comma-separated list of arguments.
- *Classname* is an optional class name that represents the type of value (if any) returned by this method. Omit the `As` *Classname* part if the method does not return a value.
- *Keywords* represents any method keywords, which control things such as how this method is projected to SQL, whether this method is available outside of the class to which it belongs, and so on. These are optional. See Compiler Keywords.
- The method implementation consists of zero or more lines of ObjectScript. This code can return a value via RETURN or QUIT.

2.11 Whitespace

Under certain circumstances, ObjectScript treats whitespace as syntactically meaningful. Unless otherwise specified, whitespace refers to blank spaces, tabs, and line feeds interchangeably. In brief, the rules are:

- Whitespace must appear at the beginning of each line of code and each single-line comment. Leading whitespace is *not* required for the following kinds of items:
  - *Label* (also known as a *tag* or an *entry point*): a label must appear in column 1 with no preceding whitespace character. If a line has a label, there must be whitespace between the label and any code or comment on the same line. If a label has a parameter list, there can be no whitespace between the label name and the opening parenthesis for the parameter list. There can be whitespace before, between, or after the parameters in the parameter list.
  - *Macro directive*: a macro directive such as #define can appear in column 1 with no preceding whitespace character. This is a recommended convention, but whitespace is permitted before a macro directive.
  - Multiline comment: the first line of a multiline comment must be preceded by one or more spaces. The second and subsequent lines of a multiline comment do not require leading whitespace.
  - Blank line: if a line contains no characters, it does not need to contain any spaces. A line consisting only of whitespace characters is permitted and treated as a comment.
- There must be one and only one space (not a tab) between a command and its first argument; if a command uses a postconditional, there are no spaces between the command and its postconditional.
- If a postconditional expression includes any spaces, then the entire expression must be parenthesized.
- There can be any amount of whitespace between any pair of command arguments.
- If a line contains code and then a single-line comment, there must be whitespace between them.
- Typically, each command appears on its own line, though you can enter multiple commands on the same line. In this case, there must be whitespace between them; if a command is argumentless, then it must be followed by two spaces (two spaces, two tabs, or one of each). Additional whitespace may follow these two required spaces.

2.12 Labels

Any line of ObjectScript code can optionally include a label (also known as a tag). A label serves as a handle for referring to that line location in the code. A label is an identifier that is not indented; it is specified in column 1. All ObjectScript commands must be indented.

Labels have the following naming conventions:

- The first character must be an alphanumeric character or the percent character (%). Note that labels are the only ObjectScript names that can begin with a number. The second and all subsequent characters must be alphanumeric characters. A label may contain Unicode letters.
- They can be up to 31 characters long. A label may be longer than 31 characters, but must be unique within the first 31 characters. A label reference matches only the first 31 characters of the label. However, all characters of a label or label reference (not just the first 31 characters) must abide by label character naming conventions.
- They are case-sensitive.

Note: A block of ObjectScript code specified in an SQL command such as CREATE PROCEDURE or CREATE TRIGGER can contain labels. In this case, the first character of the label is prefixed by a colon (:) specified in column 1. The rest of the label follows the naming and usage requirements describe here.

A label can include or omit parameter parentheses. If included, these parentheses may be empty or may include one or more comma-separated parameter names. A label with parentheses identifies a procedure block.

A line can consist of only a label, a label followed by one or more commands, or a label followed by a comment. If a command or a comment follows the label on the same line, they must be separated from the label by a space or tab character.

The following are all unique labels:

ObjectScript

```
maximum
Max
MAX
%control
```

You can use the $ZNAME function to validate a label name. Do not include parameter parentheses when validating a label name.

Labels are useful for identifying sections of code and for managing flow of control. See Legacy Code and Labels.

2.13 Comments

It is good practice to use *comments* to provide in-line documentation in code, as they are a valuable resource when modifying or maintaining code. ObjectScript supports several types of comments which can appear in several kinds of locations:

- Comments in INT Code for Routines and Methods
- Comments in MAC Code for Routines and Methods
- Comments in Class Definitions Outside of Method Code

2.13.1 Comments in INT Code for Routines and Methods

ObjectScript code is written as MAC code, from which INT (intermediate) code is generated. Comments written in MAC code are generally available in the corresponding INT code. You can use the ZLOAD command to load an INT code routine, then use the ZPRINT command or the $TEXT function to display INT code, including these comments. The following types of comments are available, all of which must start in column 2 or greater:

- The `/* */` multiline comment can appear within a line or across lines. `/*` can be the first element on a line or can follow other elements; `*/` can be the final element on the line or can precede other elements. All lines in a `/* */` appear in the INT code, including lines that consist of just the `/*` or `*/`, with the exception of completely blank lines. A blank line within a multi-line comment is omitted from the INT code, and can thus affect the line count.
- The `//` comment specifies that the remainder of the line is a comment; it can be the first element on the line or follow other elements.
- The `;` comment specifies that the remainder of the line is a comment; it can be the first element on the line or can follow other elements.
- The `;;` comment — a special case of the `;` comment type — makes the comment available to the $TEXT function when the routine is distributed as object code only; the comment is only available to **$TEXT** if no commands precede it on the line.

Note: Because InterSystems IRIS retains `;;` comments in the object code (the code that is actually interpreted and executed), there is a performance penalty for including them and they should not appear in loops.

A multiline comment (`/* comment */`) can be placed between command or function arguments, either before or after a comma separator. A multiline comment cannot be placed within an argument, or be placed between a command keyword and its first argument or a function keyword and its opening parenthesis. It can be placed between two commands on the same line, in which case it functions as the single space needed to separate the commands. You can immediately follow the end of a multiline comment (*/) with a command on the same line, or follow it with a single line comment on the same line. The following example shows these insertions of `/* comment */` within a line:

ObjectScript

```
WRITE $PIECE("Fred&Ginger"/* WRITE "world" */,"&",2),!
WRITE "hello",/* WRITE "world" */" sailor",!
SET x="Fred"/* WRITE "world" */WRITE x,!
WRITE "hello"/* WRITE "world" *//// WRITE " sailor"
```

2.13.2 Comments in MAC Code for Routines and Methods

The following comment types can be written in MAC code but have different behaviors in the corresponding INT code:

- The `#;` comment can start in any column but must be the first element on the line. #: comments do not appear in INT code. Neither the comment nor the comment marker (#;) appear in the INT code and no blank line is retained. Therefore, the `#;` comment can change INT code line numbering.
- The `##;` comment can start in any column. It can be the first element on the line or can follow other elements. `##;` comments do not appear in INT code. ##: can be used in ObjectScript code, in Embedded SQL code, or on the same line as a #define, #def1arg or ##continue macro preprocessor directive.

  If the `##;` comment starts in column 1, neither the comment nor the comment marker (##;) appear in the INT code and no blank line is retained. However, if the `##;` comment starts in column 2 or greater, neither the comment nor the comment marker (##;) appear in the INT code, but a blank line is retained. In this usage, the `##;` comment does not change INT code line numbering.
- The `///` comment can start in any column but must be the first element on the line. If `///` starts in column 1, it does not appear in INT code and no blank line is retained. If `///` starts in column 2 or greater, the comment appears in INT code and is treated as if it were a `//` comment.

2.13.3 Comments in Class Definitions Outside of Method Code

Within class definitions, but outside of method definitions, several comment types are available, all of which can start in any column:

- The `//` and `/* */` comments are for comments within the class definition.
- The `///` comment serves as class reference content for the class or class member that immediately follows it. For classes themselves, the `///` comment preceding the beginning of the class definition provides the description of the class for the class reference content which is also the value of description keyword for the class). Within classes, all `///` comments immediately preceding a member (either from the beginning of the class definition or after the previous member) provide the class reference content for that member, where multiple lines of content are treated as a single block of HTML. For more information on the rules for `///` comments and the class reference, see Creating Class Documentation.

2.14 Namespaces

In InterSystems IRIS, code always runs within a namespace, which is a logical container for data and code. For information on what is available within a namespace, see Namespaces and Databases. For information on namespace names, see Configuring Namespaces.

For ObjectScript commands that accept a namespace name as an argument, you can use an implied namespace name.

In generated code, InterSystems IRIS replaces punctuation characters in explicit and implied namespace names as follows:

% = p, _ = u, – = d, @ = s, : = s, / = s, \ = s, [= s,] = s, ^ = s.

2.15 See Also

- Procedure Syntax
- Invoking Code and Passing Arguments
- ObjectScript Variables and Scope
- Macros and Include Files

3 Procedure Syntax

This topic describes the syntax for procedures, which are defined within ObjectScript routines.

3.1 Introduction

The syntax for a procedure is as follows:

```
label(argumentlist) accessmode
 {
  implementation
 }
```

Or:

```
label(argumentlist) [pubvarlist] accessmode
 {
  implementation
 }
```

Where:

label

The procedure name, a standard label. It must start in column one. The parentheses following the label are mandatory, even if there are no arguments.

argumentlist

A comma-separated list of arguments in the following form:

```
argument1,argument2,argument3,...
```

An argument can have a default value, as follows:

```
argument1=default1,argument2=default2,argument3=default3,...
```

These expected arguments are known as the *formal arguments list*. Even in a case when the procedure takes no arguments, the procedure definition must include parentheses. The maximum number of formal parameters is 255.

Any default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string (`""`) as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, InterSystems IRIS issues a <PARAMETER> error./

pubvar

Public variables. An optional comma-separated list of public variables used by the procedure and available to other routines and procedures. This is a list of variables both defined within this procedure and available to other routines and defined within another routine and available to this procedure. If specified, *pubvar* is enclosed in square brackets. If no *pubvar* is specified, the square brackets may be omitted. The public variables can include arguments specified for this procedure.

Note: Most procedures do not declare a list of public variables. In newer code, a list of public variables is a less commonly used feature of ObjectScript.

accessmode

An optional keyword that controls how this procedure can be used. This must be one of the following:

- `PUBLIC`, which declares that this procedure can be called from any routine.
- `PRIVATE`, which declares that this procedure can only be called from the routine in which it is defined. `PRIVATE` is the default.

For instance the following defines a public procedure:

ObjectScript

```
MyProc(x,y) PUBLIC { }
```

In contrast, the following examples define a private procedures:

ObjectScript

```
MyProc(x,y) PRIVATE { }
```

```
MyProc2(x,y) { }
```

implementation

ObjectScript commands. The opening curly brace (`{`) must be separated from the characters preceding and following it by at least one space or a line break. The closing curly brace (`}`) must not be followed by any code on the same line; it can only be followed by blank space or a comment. The closing curly brace can be placed in column one. This block of code is only entered by the label.

3.2 Procedures as Functions

If an ObjectScript procedure returns a value, it is also called a *function*. Such a function is also sometimes called an *extrinsic function*, to distinguish it from built-in ObjectScript functions (which are sometimes called *intrinsic functions*).

To return a value from a procedure, use the RETURN command.

3.3 Procedure Variables

Procedures and methods both support private and public variables; all of the following statements apply equally to procedures and methods:

Variables used within procedures are automatically *private* to that procedure. To share some of these variables with procedures that this procedure calls, pass them as parameters to the other procedures.

3.3.1 Public Variable List

Via the public variable list, you can also declare *public* variables. These are available to all procedures and methods; those that this procedure or method calls and those that called this procedure or method. A relatively small number of variables should be defined in this way, to act as environmental variables for an application. To define public variables, list them in square brackets following the procedure name and its parameters.

The following example defines a procedure with two declared public variables [a, b] and two private variables (c, d):

ObjectScript

```
publicvarsexample
    ; examples of public variables
    ;
    DO proc1()   ; call a procedure
    QUIT    ; end of the main routine
    ;
proc1() [a, b]
    ; a private procedure
    ; "c" and "d" are private variables
    {
    WRITE !, "setting a"  SET a = 1
    WRITE !, "setting b"  SET b = 2
    WRITE !, "setting c"  SET c = 3
    SET d = a + b + c
    WRITE !, "The sum is: ", d
    }
```

Terminal

```
USER>WRITE

USER>DO ^publicvarsexample

setting a
setting b
setting c
The sum is: 6
USER>WRITE

a=1
b=2
USER>
```

3.3.2 Making Formal List Parameters Public

If a procedure has a formal list parameter, (such as x or y in **MyProc(x,y)**) that is needed by other procedures it calls, then the parameter should be listed in the public list.

Thus,

ObjectScript

```
MyProc(x,y)[x] {
 DO abc^rou
 }
```

makes the value of *x*, but not *y*, available to the routine abc^rou.

3.3.3 Private Variables versus Variables Created with NEW

Note that private variables are not the same as variables newly created with **NEW**. If a procedure wants to make a variable directly available to other procedures or subroutines that it calls, then it must be a public variable and it must be listed in the public list. If it is a public variable being introduced by this procedure, then it makes sense to perform a **NEW** on it. That way it will be automatically destroyed when the procedure exits, and also it protects any previous value that public variable may have had. For example, the code:

ObjectScript

```
MyProc(x,y)[name]{
 NEW name
 SET name="John"
 DO xyz^abc
}
```

enables procedure `xyz` in routine `abc` to see the value John for *name*, because it is public. Invoking the **NEW** command for *name* protects any public variable named name that may already have existed when the procedure `MyProc` was called.

The **NEW** command does not affect private variables; it only works on public variables. Within a procedure, it is illegal to specify `NEW x` or `NEW (x)` if *x* is not listed in the public list and *x* is not a % variable.

3.4 Procedure Code

The body of code between the braces is the procedure code, and it differs from traditional ObjectScript code in the following ways:

- A procedure can only be entered at the procedure label. Access to the procedure through *label+offset* syntax is not allowed.
- Any labels in the procedure are private to the procedure and can only be accessed from within the procedure. The PRIVATE keyword can be used on labels within a procedure, although it is not required. The PUBLIC keyword cannot be used on labels within a procedure — it yields a syntax error. Even the system function **$TEXT** cannot access a private label by name, although **$TEXT** does support *label+offset* using the procedure label name.
- Duplicate labels are not permitted within a procedure but, under certain circumstances, are permitted within a routine. Specifically, duplicate labels are permitted within different procedures. Also, the same label can appear within a procedure and elsewhere within the routine in which the procedure is defined. For instance, the following three occurrences of `Label1` are permitted:

  ObjectScript

  ```
  Rou1 // Rou1 routine
  Proc1(x,y) {
  Label1 // Label1 within the proc1 procedure within the Rou1 routine
  }

  Proc2(a,b,c) {
  Label1 // Label1 within the Proc2 procedure (local, as with previous Label1)
  }

  Label1 // Label1 that is part of Rou1 and neither procedure
  ```

- If the procedure contains a **DO** command or user-defined function without a routine name, it refers to a label within the procedure, if one exists. Otherwise, it refers to a label in the routine but outside of the procedure.

- If the procedure contains a **DO** or user-defined function with a routine name, it always identifies a line outside of the procedure. This is true even if that name identifies the routine that contains the procedure. For example:

  ObjectScript

  ```
  ROU1 ;
  PROC1(x,y) {
   DO Label1^ROU1
  Label1 ;
   }
  Label1 ; The DO calls this label
  ```

- If a procedure contains a **GOTO**, it must be to a private label within the procedure. You cannot exit a procedure with a **GOTO**.
- *label+offset* syntax is not supported within a procedure, with a few exceptions:
  - **$TEXT** supports *label+offset* from the procedure label.
  - **GOTO** *label+offset* is supported in direct mode lines from the procedure label as a means of returning to the procedure following a **Break** or error.
  - The **ZBREAK** command supports a specification of *label+offset* from the procedure label.
- When the procedure ends, the system restores the **$TEST** state that had been in effect when the procedure was called.
- The `}` that denotes the end of the procedure can be in any character position on the line, including the first character position. Code can precede the `}` on the line, but cannot follow it on the line.
- An implicit **QUIT** is present just before the closing brace.
- Indirection and **XECUTE** commands behave as if they are outside of a procedure.

3.5 Indirection, XECUTE Commands, and JOB Commands within Procedures

Name indirection, argument indirection, and **XECUTE** commands that appear within a procedure are not executed within the scope of the procedure. Thus, **XECUTE** acts like an implied **DO** of a subroutine that is outside of the procedure.

Indirection and **XECUTE** only access public variables. As a result, if indirection or an **XECUTE** references a variable *x*, then it references the public variable *x* regardless of whether or not there is also a private *x* in the procedure. For example:

ObjectScript

```
 SET x="set a=3" XECUTE x ; sets the public variable a to 3
 SET x="label1" DO @x ; accesses the public subroutine label1
```

Similarly, a reference to a label within indirection or an **XECUTE** is to a label outside of the procedure. Hence `GOTO @A` is not supported within a procedure, since a **GOTO** from within a procedure must be to a label within the procedure.

Other parts of the documentation contain more detail on indirection and the XECUTE command.

Similarly, when you issue a JOB command within a procedure, it starts a child process that is outside the method. This means that for code such as the following:

ObjectScript

```
    KILL ^MyVar
    JOB MyLabel
    QUIT $$$OK
MyLabel
    SET ^MyVar=1
    QUIT
```

In order for the child process to be able to see the label, the method or the class cannot be contained in a procedure block.

3.6 Error Traps within Procedures

If an error trap gets set from within a procedure, it needs to be directly to a private label in the procedure. (This is unlike in legacy code, where it can contain +*offset* or a routine name. This rule is consistent with the idea that executing an error trap essentially means unwinding the stack back to the error trap and then executing a **GOTO**.)

If an error occurs inside a procedure, $ZERROR gets set to the procedure *label+offset*, not to a private *label+offset*.

To set an error trap, the normal **$ZTRAP** is used, but the value must be a literal. For instance:

ObjectScript

```
 SET $ZTRAP = "abc"
 // sets the error trap to the private label "abc" within this block
```

For more information on error traps, see Using Try-Catch.

4

Introduction to ObjectScript Commands

This topic introduces some of the most commonly used ObjectScript commands; also see the *ObjectScript Reference*.

4.1 Command to Assign Values

Use the **SET** command to assign a value to a variable. The basic syntax of **SET** is:

ObjectScript

```
SET MyVar=expression
```

where *MyVar* is a variable and expression is any ObjectScript expression that is suitable in the given context. See ObjectScript Variables.

4.2 Commands to Invoke Code

This section describes the commands used for invoking code:

- DO
- JOB
- XECUTE
- QUIT and RETURN

Also see the page Invoking Code and Passing Arguments.

4.2.1 DO

To invoke a routine, procedure, or method in ObjectScript, use the **DO** command. The basic syntax of **DO** is:

ObjectScript

```
DO ^CodeToInvoke
```

where *CodeToInvoke* can be an InterSystems IRIS system routine or a user-defined routine. The caret character ^ must appear immediately before the name of the routine.

You can run procedures within a routine by referring to the label of the line (also called a tag) where the procedure begins within the routine. The label appears immediately before the caret. For example,

ObjectScript

```
SET %X = 484
DO INT^%SQROOT
WRITE %Y
```

This code sets the value of the *%X* system variable to 484; it then uses DO to invoke the INT procedure of the InterSystems IRIS system routine **%SQROOT**, which calculates the square root of the value in *%X* and stores it in *%Y*. The code then displays the value of *%Y* using the **WRITE** command.

When invoking methods, **DO** takes as a single argument the entire expression that specifies the method. The form of the argument depends on whether the method is an instance or a class method. To invoke a class method, use the following construction:

ObjectScript

```
DO ##class(PackageName.ClassName).ClassMethodName()
```

where **ClassMethodName()** is the name of the class method that you wish to invoke, ClassName is the name of the class containing the method, and PackageName is the name of the package containing the class. The `##class()` construction is a required literal part of the code.

To invoke an instance method, you need only have a handle to the locally instantiated object:

ObjectScript

```
DO InstanceName.InstanceMethodName()
```

where **InstanceMethodName()** is the name of the instance method that you wish to invoke, and InstanceName is the name of the instance containing the method.

For further details, see DO.

4.2.2 JOB

While **DO** runs code in the foreground, **JOB** runs it in the background. This occurs independently of the current process, usually without user interaction. A jobbed process inherits all system defaults, except those explicitly specified.

For further details, see JOB.

4.2.3 XECUTE

The **XECUTE** command runs one or more ObjectScript commands; it does this by evaluating the expression that it receives as an argument (and its argument must evaluate to a string containing one or more ObjectScript commands). In effect, each **XECUTE** argument is like a one-line subroutine called by a **DO** command and terminated when the end of the argument is reached or a **QUIT** command is encountered. After InterSystems IRIS executes the argument, it returns control to the point immediately after the **XECUTE** argument.

For further details, see XECUTE.

4.2.4 QUIT and RETURN

The **QUIT** and **RETURN** commands both terminate execution of a code block, including a method. Without an argument, they simply exit the code from which they were invoked. With an argument, they use the argument as a return value. **QUIT** exits the current context, exiting to the enclosing context. **RETURN** exits the current program to the place where the program was invoked.

The following table shows how to choose whether to use **QUIT RETURN**:

Location	QUIT	RETURN
Routine code (not block structured)	Exits routine, returns to the calling routine (if any).	Exits routine, returns to the calling routine (if any).
TRY or CATCH block	Exits TRY / CATCH block structure pair to next code in routine. If issued from a nested TRY or CATCH block, exits one level to the enclosing TRY or CATCH block.	Exits routine, returns to the calling routine (if any).
DO or XECUTE	Exits routine, returns to the calling routine (if any).	Exits routine, returns to the calling routine (if any).
IF	Exits routine, returns to the calling routine (if any). However, if nested in a FOR, WHILE, or DO WHILE loop, exits that block structure and continues with the next line after the code block.	Exits routine, returns to the calling routine (if any).
FOR, WHILE, DO WHILE	Exits the block structure and continues with the next line after the code block. If issued from a nested block, exits one level to the enclosing block.	Exits routine, returns to the calling routine (if any).

For further details, see QUIT and RETURN.

4.3 Commands to Control Flow

In order to establish the logic of any code, there must be flow control; conditional executing or bypassing blocks of code, or repeatedly executing a block of code. To that end, ObjectScript supports the following commands:

- Conditional Execution
- FOR
- WHILE and DO WHILE

4.3.1 Conditional Execution

To conditionally execute a block of code, based on boolean (true/false) test, you can use the **IF** command. (You can perform conditional execution of individual ObjectScript commands by using a postconditional expression.)

IF takes an expression as an argument and evaluates that expression as true or false. If true, then the block of code that follows the expression is executed; if false, the block of code is not executed. Most commonly these are represented by 1 and 0, which are the recommended values. However, InterSystems IRIS performs conditional execution on any value, evaluating it as False if it evaluates to 0 (zero), and True if it evaluates to a nonzero value. For further details, see Operators.

You can specify multiple **IF** boolean test expressions as a comma-separated list. These tests are evaluated in left-to-right order as a series of logical AND tests. Therefore, an **IF** evaluates as true when all of its test expressions evaluate as true. An **IF** evaluates as false when the one of its test expressions evaluates as false; the remaining test expressions are not evaluated.

The code usually appears in a *code block* containing multiple commands. Code blocks are simply one or more lines of code contained in curly braces; there can be line breaks before and within the code blocks. Consider the following:

4.3.1.1 IF, ELSEIF, and ELSE

The **IF** *construct* allows you to evaluate multiple conditions, and to specify what code is run based on the conditions. A construct, as opposed to a simple command, consists of a combination of one or more command keywords, their conditional expressions and *code blocks*. The **IF** construct consists of:

- One **IF** clause with one or more conditional expressions.
- Any number of **ELSEIF** clauses, each with one or more conditional expressions. The **ELSEIF** clause optional; there can be more than one **ELSEIF** clause.
- At most one **ELSE** clause, with no conditional expression. The **ELSE** clause is optional.

The following is an example of the **IF** construct:

ObjectScript

```
READ "Enter the number of equal-length sides in the polygon: ",x
  IF x=1 {WRITE !,"It's so far away that it looks like a point"}
  ELSEIF x=2 {WRITE !,"I think that's a line, not a polygon"}
  ELSEIF x=3 {WRITE !,"It's an equalateral triangle"}
  ELSEIF x=4 {WRITE !,"It's a square"}
  ELSE {WRITE !,"It's a polygon with ",x," number of sides" }
WRITE !,"Finished the IF test"
```

For further details, refer to the reference for IF.

4.3.2 FOR

You use the **FOR** construct to repeat sections of code. You can create a **FOR** loop based on numeric or string values.

Typically, **FOR** executes a code block zero or more times based on the value of a numeric control variable that is incremented or decremented at the beginning of each loop through the code. When the control variable reaches its end value, control exits the **FOR** loop; if there is no end value, the loop executes until it encounters a **QUIT** command. When control exits the loop, the control variable maintains its value from the last loop executed.

The form of a numeric **FOR** loop is:

ObjectScript

```
FOR ControlVariable = StartValue:IncrementAmount:EndValue {
        // code block content
}
```

All values can be positive or negative; spaces are permitted but not required around the equals sign and the colons. The code block following the **FOR** will repeat for each value assigned to the variable.

For example, the following **FOR** loop will execute five times:

ObjectScript

```
WRITE "The first five multiples of 3 are:",!
FOR multiple = 3:3:15 {
   WRITE multiple,!
}
```

You can also use a variable to determine the end value. In the example below, a variable specifies how many iterations of the loop occur:

ObjectScript

```
SET howmany = 4
WRITE "The first ",howmany," multiples of 3 are "
FOR multiple = 1:1:howmany {
    WRITE (multiple*3),", "
    IF multiple = (howmany - 1) {
        WRITE "and "
    }
    IF multiple = howmany {
        WRITE "and that's it!"
    }
}
QUIT
```

Because this example uses *multiple*, the control variable, to determine the multiples of 3, it displays the expression `multiple*3`. It also uses the **IF** command to insert and before the last multiple.

Note: The **IF** command in this example provides an excellent example of the implications of order of precedence in ObjectScript (order of precedence is always left to right with no hierarchy among operators). If the **IF** expression were simply multiple = howmany - 1, without any parentheses or parenthesized as a whole, then the first part of the expression, multiple = howmany, would be evaluated to its value of False (0); the expression as a whole would then be equal to 0 - 1, which is -1, which means that the expression will evaluate as true (and insert and for every case except the final iteration through the loop).

The argument of **FOR** can also be a variable set to a list of values; in this case, the code block will repeat for each item in the list assigned to the variable.

ObjectScript

```
FOR item = "A", "B", "C", "D" {
   WRITE !, "Now examining item: "_item
}
```

You can specify the numeric form of **FOR** without an ending value by placing a **QUIT** within the code block that triggers under particular circumstances and thereby terminates the **FOR**. This approach provides a counter of how many iterations have occurred and allows you to control the **FOR** using a condition that is not based on the counter's value. For example, the following loop uses its counter to inform the user how many guesses were made:

ObjectScript

```
   FOR i = 1:1 {
   READ !, "Capital of MA? ", a
   IF a = "Boston" {
       WRITE "...did it in ", i, " tries"
       QUIT
       }
   }
```

If you have no need for a counter, you can use the argumentless **FOR**:

ObjectScript

```
    FOR  {
        READ !, "Know what? ", wh
        QUIT:(wh = "No!")
        WRITE "   That's what!"
    }
```

For further details, see FOR.

4.3.3 WHILE and DO WHILE

Two related flow control commands are **WHILE** and **DO WHILE** commands, each of which loops over a code block and terminates based on a condition. The two commands differ in when they evaluate the condition: **WHILE** evaluates the condition before the entire code block and **DO WHILE** evaluates the condition after the block. As with **FOR**, a **QUIT** within the code block terminates the loop.

The syntax for the two commands is:

```
 DO {code} WHILE condition
 WHILE condition {code}
```

The following example displays values in the Fibonacci sequence up to a user-specified value twice — first using **DO WHILE** and then using **WHILE**:

ObjectScript

```
fibonacci() PUBLIC { // generate Fibonacci sequences
    READ !, "Generate Fibonacci sequence up to where? ", upto
    SET t1 = 1, t2 = 1, fib = 1
    WRITE !
    DO {
        WRITE fib,"  "  set fib = t1 + t2, t1 = t2, t2 = fib
    }
    WHILE ( fib '> upto )

    SET t1 = 1, t2 = 1, fib = 1
    WRITE !
    WHILE ( fib '> upto ) {
        WRITE fib,"  "
        SET fib = t1 + t2, t1 = t2, t2 = fib
    }
 }
```

The distinction between **WHILE**, **DO WHILE**, and **FOR** is that **WHILE** necessarily tests the control expression's value before executing the loop, **DO WHILE** necessarily tests the value after executing the loop, and **FOR** can test it anywhere within the loop. This means that if you have two parts to a code block, where execution of the second depends on evaluating the expression, the **FOR** construct is best suited; otherwise, the choice depends on whether expression evaluation should precede or follow the code block.

For further details, see WHILE and DO WHILE.

4.4 Commands to Processes Error

Use the **TRY** / **CATCH** block structure for error processing: It is recommended that you use the **TRY** and **CATCH** commands to create block structures for error processing.

See The TRY-CATCH Mechanism, and see TRY, THROW, and CATCH.

4.5 Commands to Process Transactions

Use the **TSTART**, **TCOMMIT**, and **TROLLBACK** commands for transaction processing. See Transaction Processing, and see TSTART, TCOMMIT, and TROLLBACK.

4.6 Command for Locking and Concurrency Control

Use the **LOCK** command for locking and unlocking resources. See Locking and Concurrency Control and see LOCK.

Locking is also relevant in transaction processing; see Transaction Processing.

4.7 Write Commands

ObjectScript supports four commands to display (write) literals and variable values to the current output device:

- WRITE command
- ZWRITE command
- ZZDUMP command
- ZZWRITE command

4.7.1 Argumentless Display Commands

- Argumentless **WRITE** displays the name and value of each defined local variable, one variable per line. It lists both public and private variables. It does not list global variables, process-private globals, or special variables. It lists variables in collation sequence order. It lists subscripted variables in subscript tree order.

  It displays all data values as quoted strings delimited by double quote characters, except for canonical numbers and object references. It displays a variable assigned an object reference (OREF) value as `variable=<OBJECT REFERENCE>[oref]`. It displays a %List format value or a bitstring value in their encoded form as a quoted string. Because these encoded forms may contain non-printing characters, a %List or bitstring may appear to be an empty string.

  WRITE does not display certain non-printing characters; no placeholder or space is displayed to represent these non-printing characters. **WRITE** executes control characters (such as line feed or backspace).
- Argumentless **ZWRITE** is functionally identical to argumentless **WRITE**.
- Argumentless **ZZDUMP** is an invalid command that generates a <SYNTAX> error.
- Argumentless **ZZWRITE** is a no-op that returns the empty string.

4.7.2 Display Commands with Arguments

The following tables list the features of the argumented forms of the four commands. All four commands can take a single argument or a comma-separated list of arguments. All four commands can take as an argument a local, global, or process-private variable, a literal, an expression, or a special variable:

The following tables also list the **%Library.Utility.FormatString()** method default return values. The **FormatString()** method is most similar to **ZZWRITE**, except that it does not list `%val=` as part of the return value, and it returns only the object reference (OREF) identifier. **FormatString()** allows you to set a variable to a return value in **ZWRITE** / **ZZWRITE** format.

Table 4–1: Display Formatting

	WRITE	**ZWRITE**	**ZZDUMP**	**ZZWRITE**	**FormatString()**
Each value on a separate line?	NO	YES	YES (16 characters per line)	YES	One input value only
Variable names identified?	NO	YES	NO	Represented by `%val=`	NO
Undefined variable results in <UNDEFINED> error?	YES	NO (skipped, variable name not returned)	YES	YES	YES

All four commands evaluate expressions and return numbers in canonical form.

Table 4–2: How Values are Displayed

	WRITE	ZWRITE	ZZDUMP	ZZWRITE	FormatString()

	WRITE	**ZWRITE**	**ZZDUMP**	**ZZWRITE**	**FormatString()**
Hexadecimal representation?	NO	NO	YES	NO	NO
Strings quoted to distinguish from numerics?	NO	YES	NO	YES (a string literal is returned as `%val="value"`)	YES
Subscript nodes displayed?	NO	YES	NO	NO	NO
Global variables in another namespace (extended global reference) displayed?	YES	YES (extended global reference syntax shown)	YES	YES	YES
Non-printing characters displayed?	NO, not displayed; control characters executed	YES, displayed as `$c(n)`	YES, displayed as hexadecimal	YES, displayed as `$c(n)`	YES, displayed as `$c(n)`
List value format	encoded string	`$lb(val)` format	encoded string	`$lb(val)` format	`$lb(val)` format
%Status format	string containing encoded Lists	string containing `$lb(val)` format Lists, with appended /*... */ comment specifying error and message.	string containing encoded Lists	string containing `$lb(val)` format Lists, with appended /*... */ comment specifying error and message.	string containing `$lb(val)` format Lists, with (by default) appended /*... */ comment specifying error and message.
Bitstring format	encoded string	$zwc format with appended /* $bit() */ comment listing 1 bits. For example: %val=$zwc(407,2,1,2,3,5)/*$bit(2..4,6)*/	encoded string	$zwc format with appended /* $bit() */ comment listing 1 bits. For example: %val=$zwc(407,2,1,2,3,5)/*$bit(2..4,6)*/	$zwc format with (by default) appended /* $bit() */ comment listing 1 bits. For example: %val=$zwc(407,2,1,2,3,5)/*$bit(2..4,6)*/

	WRITE	ZWRITE	ZZDUMP	ZZWRITE	FormatString()
Object Reference (OREF) format	OREF only	OREF in <OBJECT REFERENCE>[oref] format. General information, attribute values, etc. details listed. All subnodes listed	OREF only	OREF in <OBJECT REFERENCE>[oref] format. General information, attribute values, etc. details listed.	OREF only, as quoted string

JSON dynamic arrays and JSON dynamic objects are returned as OREF values by all of these commands. To return the JSON contents, you must use **%ToJSON()**, as shown in the following example:

```
SET jobj={"name":"Fred","city":"Bedrock"}
WRITE "JSON object reference:",!
ZWRITE jobj
WRITE !!,"JSON object value:",!
ZWRITE jobj.%ToJSON()
```

For further details, see WRITE, ZWRITE, ZZDUMP, and ZZWRITE.

4.8 READ Command

The **READ** command allows you to accept and store input entered by the end user via the current input device. The **READ** command can have any of the following arguments:

ObjectScript

```
READ format, string, variable
```

Where *format* controls where the user input area will appear on the screen, *string* will appear on the screen before the input prompt, and *variable* will store the input data.

The following format codes are used to control the user input area:

Format Code	Effect
!	Starts a new line.
#	Starts a new page. On a terminal it clears the current screen and starts at the top of a new screen.
?n	Positions at the nth column position where *n* is a positive integer.

For further details, see READ.

4.9 Files and Devices

To work with files and directories, InterSystems IRIS provides the %File API.

In addition, InterSystems IRIS provides low-level commands you can use to work with devices. This process as a whole is described in the I/O Device Guide. For further details, see OPEN, USE, and CLOSE.

4.10 See Also

- ObjectScript Command Reference

5
Introduction to ObjectScript Operators

ObjectScript supports many different operators, which perform various actions, including mathematical actions, logical comparisons, and so on. Operators act on expressions, which are variables or other entities that are ultimately evaluated to a value. This topic describes expressions and the operators.

5.1 Introduction

Operators are symbolic characters that specify the action to be performed on their associated *operands*. Each operand consists of one or more *expressions* or *expression atoms*. When used together, an operator and its associated operands have the following form:

[*operand*] operator *operand*

Some operators take only one operand and are known as unary operators; others take two operands and are known as binary operators.

An operator and any of its operands taken together constitute an expression.

5.1.1 Assignment

Within ObjectScript the SET command is used along with the assignment operator (=) to assign a value to a variable. The right-hand side of an assignment command is an expression:

ObjectScript

```
SET value = 0
SET value = a + b
```

Within ObjectScript it is also possible to use certain functions on the left-hand side of an assignment command:

ObjectScript

```
SET pies = "apple,banana,cherry"
WRITE "Before: ",pies,!

// set the 3rd comma-delimited piece of pies to coconut
SET $Piece(pies,",",3) = "coconut"
WRITE "After: ",pies
```

5.2 Operator Precedence

Operator precedence in ObjectScript is strictly left-to-right; within an expression operations are performed in the order in which they appear. This is different from other languages in which certain operators have higher precedence than others. You can use explicit parentheses within an expression to force certain operations to be carried ahead of others. For example:

Terminal

```
USER>WRITE "1 + 2 * 3 = ", 1 + 2 * 3
1 + 2 * 3 = 9
USER>WRITE 1 + 2 * 3
9
USER>WRITE 2 * 3 + 1
7
USER>WRITE 1 + (2 * 3)
7
USER>WRITE 2 * (3 + 1)
8
```

Note that in InterSystems SQL, operator precedence is configurable, and may (or may not) match the operator precedence in ObjectScript.

5.2.1 Unary Negative Operators

ObjectScript gives the unary negative operator precedence over the binary arithmetic operators. ObjectScript first scans a numeric expression and performs any unary negative operations. Then, ObjectScript evaluates the expression and produces a result. For example:

Terminal

```
USER>WRITE -123 - 3
-126
USER>WRITE -123 + - 3
-126
USER>WRITE -(123 - 3)
-120
```

5.2.2 Parentheses and Precedence

You can change the order of evaluation by nesting expressions within each other with matching parentheses. The parentheses group the enclosed expressions (both arithmetic and relational) and control the order of operations. For example:

Terminal

```
USER>SET TorF = ((4 + 7) > (6 + 6))

USER>WRITE TorF
0
```

Here, because of the parentheses, four and seven are added, as are six and six; this results in the logical expression `11 > 12`, which is false. Compare this to:

Terminal

```
USER>SET Value = (4 + 7 > 6 + 6)

USER>WRITE Value
7
```

In this case, precedence proceeds from left to right, so four and seven are added. Their sum, eleven, is compared to six; since eleven is greater than six, the result of this logical operation is one (TRUE). One is then added to six, and the result is seven.

Note that the precedence even determines the result type, since the first expression's final operation results in a boolean and the second expression's final operation results in a numeric.

The following example shows multiple levels of nesting:

Terminal

```
USER>WRITE 1+2*3-4*5
25
USER>WRITE 1+(2*3)-4*5
15
USER>WRITE 1+(2*(3-4))*5
-5
USER>WRITE 1+(((2*3)-4)*5)
11
```

Precedence from the innermost nested expression and proceeds out level by level, evaluating left to right at each level.

Tip: For all but the simplest ObjectScript expressions, it is good practice to fully parenthesize expressions. This is to eliminate any ambiguity about the order of evaluation and to also eliminate any future questions about the original intention of the code.

For example, because the && operator, like all operators, is subject to left-to-right precedence, the final statement in the following code fragment evaluates to 0:

Terminal

```
USER>SET x = 3

USER>SET y = 2

USER>WRITE x && y = 2
0
```

This is because, within the last step, the evaluation occurs as follows:

1. The first action is to check if *x* is defined and has a non-zero value. Since *x* equals 3, evaluation continues.
2. Next, there is a check if *y* is defined and has a non-zero value. Since *y* equals 2, evaluation continues.
3. Next, the value of `3 && 2` is evaluated. Since neither 3 nor 2 equal 0, this expression is true and evaluates to 1.
4. The next action is to compare the returned value to 2. Since 1 does not equal 2, this evaluation returns 0.

For those accustomed to many programming languages, this is an unexpected result. If the intent is to return True if *x* is defined with a non-zero value and if *y* equals 2, then parentheses are required:

Terminal

```
USER>SET x = 3

USER>SET y = 2

USER>WRITE x && (y = 2)
1
```

5.2.3 Functions and Precedence

Some types of expressions, such as functions, can have side effects. Suppose you have the following logical expression:

ObjectScript

```
 IF var1 = ($$ONE + (var2 * 5)) {
    DO ^Test
 }
```

ObjectScript first evaluates *var1*, then the function **$$ONE**, then *var2*. It then multiplies *var2* by 5. Finally, ObjectScript tests to see if the result of the addition is equal to the value in *var1*. If it is, it executes the **DO** command to call the **Test** routine.

As another example, consider the following example:

Terminal

```
USER>SET var8=25,var7=23

USER>WRITE var8 = 25 * (var7 < 24)
1
```

ObjectScript evaluates expressions strictly left-to-right. The programmer must use parentheses to establish any precedence. In this case, ObjectScript first evaluates the logical expression `var8 = 25`, resulting in 1. It then multiplies this result with the results of `(var7 < 24)`. The expression `(var7 < 24)` evaluates to 1. Therefore, ObjectScript multiplies 1 by 1, resulting in 1.

5.3 Numeric Operators

You can use the Equals operator (=) to test for numeric equality if both operands have a numeric value. Other ObjectScript operators interpret their operands as numeric values and can be used only when the operands can be interpreted as numbers. These operators are as follows:

- `+`
- `-`
- `*`
- `/`
- `\`
- `#`
- `**`
- `<`
- `>`
- `<=`
- `>=`

See Numeric Values in ObjectScript.

5.4 String Operators

With the Equals operator (=), if the operands cannot be interpreted as numbers, the operator tests for string equality.

Other ObjectScript operators always interpret their operands as strings. These operators are as follows:

- _ (the concatenate operator)
- [(the contains operator)
-] (the follows operator)
-]] (the sorts after operator)
- ? (the pattern match operator)

See Strings in ObjectScript.

5.5 Boolean Operators

ObjectScript provides operators that always interpret their operands as logical values. These are as follows:

- ' (logical NOT)
- & and && (logical AND)
- ! and || (logical OR)

Unlike some other languages, ObjectScript does not provide specialized representations of Boolean literal values. Instead, any expression that be interpreted as a nonzero numeric value is considered true; any other expression is false. See String-to-Number Conversions.

See Boolean Values in ObjectScript.

5.6 Indirection Operator (@)

Indirection is a technique that provides dynamic runtime substitution of part or all of a command line, a command, or a command argument by the contents of a data field.

Indirection is specified by the indirection operator (@) and, except for subscript indirection, takes the form:

@variable

where *variable* identifies the variable from which the substitution value is to be taken. All variables referenced in the substitution value are public variables, even when used in a procedure. The variable can be an array node.

The following routine illustrates that indirection looks at the entire variable value to its right.

ObjectScript

```
IndirectionExample
 SET x = "ProcA"
 SET x(3) = "ProcB"
 ; The next line will do ProcB, NOT ProcA(3)
 DO @x(3)
 QUIT
ProcA(var)
 WRITE !,"At ProcA"
 QUIT
ProcB(var)
 WRITE !,"At ProcB"
 QUIT
```

For details, see the Indirection (@) reference page.

Note: Although indirection can promote more economical and more generalized coding than would be otherwise available, it is never essential. You can always duplicate the effect of indirection by other means, such as by using the XECUTE command.

5.7 See Also

- Numeric Values in ObjectScript, which includes String-to-Number Conversions
- Strings in ObjectScript
- Boolean Values in ObjectScript
- ObjectScript Operator Reference

6

Invoking Code and Passing Arguments

This page describes how to invoke units of code and pass arguments to that code.

For an introduction to the commands shown here, see Commands to Invoke Code.

6.1 Calling Units of Code

The syntax to call a unit of code depends on the type of code you are calling, as well as whether you are obtaining any value returned by that code:

ObjectScript function

To call an ObjectScript function, write an expression of the following form, including that expression wherever the resulting value is needed:

```
$functionName(args)
```

For example:

```
set myvariable=$length("this is a sample string")
```

routines

To call a routine, use the DO command or one of the other commands to invoke code, with syntax that refers to the routine name:

```
DO ^routinename
```

```
JOB ^routinename
```

procedure

To call a procedure, use either the DO command or one of the other commands to invoke code, with syntax that refers to the procedure name:

```
DO procedure^routinename
```

```
JOB procedure^routinename
```

You can omit the `^routinename` part if this command is within the same routine.

If the procedure accepts arguments, you can include the argument list at the end, for example:

```
 DO procedure^routinename(argument1,argument2)
```

Depending on the procedure implementation, it can return a value. If so, and if you want to obtain the value, write an expression of the following form:

```
$$procedure^routinename
```

For example:

```
 set myvariable=$$procedure^routinename
```

You can omit the `^routinename` part if this command is within the same routine.

As before, if the procedure accepts arguments, you can include the argument list at the end, for example:

```
$$procedure^routinename(argument1,argument2)
```

If you attempt to access a private procedure from outside the routine that defines it, a <NOLINE> error occurs.

methods

To invoke a class method, use an expression of the following form:

```
##class(Package.Class).MethodName(args)
```

For example:

```
do ##class(Package.Class).MethodName(args)
```

Or:

```
set myvariable=##class(Package.Class).MethodName(args)
```

Unlike with procedures, you must include the trailing parentheses even if the method takes no arguments.

To invoke an instance method, you first need an OREF that contains a reference to the relevant object. For example:

```
 set oref=##class(Sample.Class).%OpenId(10000)
 do oref.WriteAddress()
```

Or:

```
 set oref=##class(Sample.Class).%OpenId(10000)
 set myvariable=oref.WriteAddress()
```

For details and variations, see Working with Registered Objects.

6.2 Formal Argument Lists and Examples

Each ObjectScript function, each procedure (or other form of subroutine), and each method has (or can have) a formal argument list, which is a comma-separated list of arguments. This section presents basic examples.

ObjectScript functions

For an ObjectScript function, the corresponding reference page shows the formal argument list. For example, the reference page for the **$LENGTH** function shows the formal argument list as follows:

```
$LENGTH(expression,delimiter)
```

Methods

For methods, the class documentation shows the formal argument list and any generated comments, and you can also view the code directly in your IDE. In contrast to the ObjectScript function documentation, the formal argument list includes information about the expected values of the arguments and default values.

For example, %Library.File shows the formal argument list for the **NormalizeFilename()** method as follows:

```
classmethod NormalizeFilename(filename As %String, directory As %String = "") as %String
```

procedures and other forms of subroutines

For any type of subroutine explained in the documentation, the documentation generally shows the formal argument list in context—for example as part of the command you would use. For example, the reference page for the ^PERFMON routine shows the syntax for calling the **Collect()** function within this routine, as follows:

```
 set status = $$Collect^PERFMON(time,format,output)
```

For any new routines, InterSystems suggests you create procedures, but you may encounter legacy forms of subroutines.

6.3 Passing Arguments (Basics)

When you invoke a unit of code, you pass arguments, usually using syntax of the following general form:

```
 codeunitidentifier(arg1,arg2,arg3)
```

Note the following general points:

- This syntax passes arguments *by value*. An alternative, less common-form passes arguments by reference.
- Some units of code do not accept any arguments.

  If the unit of code is a method or an ObjectScript function, you must include the parentheses after its name. If the unit of code is a subroutine (of any form), you do not need to include the parentheses after its name.
- InterSystems IRIS maps the given arguments, by position, to the corresponding arguments in the formal argument list. Thus, the value of the first arguments in the actual list is placed in the first variable in the formal list; the second value is placed in the second variable; and so on.

  The matching of these arguments is done by position, not name.
- In some cases, some arguments are optional. If an argument is optional, and you need to specify an argument in a later position, simply use commas to skip the arguments that you do not want to pass. For example:

  ```
   set myval=##class(Sample.MyClass).MyMethod(arg1,,,arg4)
  ```
- If you pass more arguments than are present in the formal argument list, a <PARAMETER> error occurs. (This does not include code units that accept a variable number of arguments; these never produce a <PARAMETER> error.)

Variations are discussed in the following sections.

6.4 Passing ByRef or Output Arguments

Some argument lists include the keyword `ByRef` or the keyword `Output` before one or more arguments. For example:

- %Library.Persistent shows the formal argument list for the **%OpenID()** method as follows, with line breaks for readability:

  ```
  classmethod %OpenID(id As %String="",
                      concurrency As %Integer = -1,
                      ByRef sc As %Status = $$$OK) as %ObjectHandle
  ```

- %Library.File shows the formal argument list for the **Exists()** method as follows:

  ```
  classmethod Exists(filename As %String, Output return As %Integer) as %Boolean
  ```

In these cases, when calling such units of code, place a period immediately before any `ByRef` or `Output` argument. This means that the argument must be a variable, rather than a literal or other kind of expression.

In these cases, you are passing the given argument by reference. In general, this means that this argument is set by or updated by the code unit that you are calling. Similarly, this means that the argument then contains a value intended for your use, such as in deciding how to proceed.

For example, when calling the **Exists()** method of the %Library.File class, use a period before the second argument:

```
 set status=##class(%Library.File).Exists("c:\temp\check.txt",.returncode)
```

As indicated in the class reference, the second argument contains the value obtained from the operating system when this check is performed.

Similarly, when calling the **%OpenID()** method of %Library.Persistent, inherited by all persistent classes, use a period before the third argument:

```
 set myvar=##class(MyPackage.MyClass).%OpenId(10034,,.statuscode)
```

As indicated in the class reference, the third argument contains the status code that indicates success or failure (and the reason, in case of failure).

Except for ObjectScript functions, any unit of code can be written to accept arguments passed by reference.

For information on defining a method this way, see Indicating How Arguments Are to Be Passed.

6.5 Comparison: Arguments by Value and Arguments by Reference

This section describes the differences between the two ways of passing arguments. This section first discusses passing local variables with no subscripts (the most common scenario).

As with other programming languages, InterSystems IRIS has a memory location that contains the value of each local variable. The name of the variable acts as the address to the memory location.

When you pass a local variable with no subscripts to a method, you pass the variable *by value*. This means that the system makes a copy of the value, so that the original value is not affected. To pass the memory address instead, place a period immediately before the name of the variable in the argument list.

To demonstrate this, consider the following method in a class called **Test.Parameters**:

Class Member

```
ClassMethod Square(input As %Integer) As %Integer
{
    set answer=input*input
    set input=input + 10
    return answer
}
```

Suppose that you define a variable and pass it by value to this method:

```
TESTNAMESPACE>set myVariable = 5

TESTNAMESPACE>write ##class(Test.Parameters).Square(myVariable)
25
TESTNAMESPACE>write myVariable
5
```

In contrast, suppose that you pass the variable by reference:

```
TESTNAMESPACE>set myVariable = 5

TESTNAMESPACE>write ##class(Test.Parameters).Square(.myVariable)
25
TESTNAMESPACE>write myVariable
15
```

Consider the following method, which writes the contents of the argument it receives:

```
ClassMethod WriteContents(input As %String)
{
    zwrite input
}
```

Now, suppose you have an array with three nodes:

```
TESTNAMESPACE>zwrite myArray
myArray="Hello"
myArray(1)="My"
myArray(2)="Friend"
```

If you pass the array to the method by value, you are only passing the top-level node:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(myArray)
input="Hello"
```

If you pass the array to the method by reference, you are passing the entire array:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(.myArray)
input="Hello"
input(1)="My"
input(2)="Friend"
```

You can pass the value of a single node of a global to a method:

```
TESTNAMESPACE>zwrite ^myGlobal
^myGlobal="Start"
^myGlobal(1)="Your"
^myGlobal(2)="Engines"
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(^myGlobal)
input="Start"
```

Trying to pass a global to a method by reference results in a syntax error:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(.^myGlobal)
^
<SYNTAX>
```

The following table summarizes all the possibilities:

Kind of Variable	Passing by Value	Passing by Reference
Local variable with no subscripts	The standard way in which these variables are passed	Allowed
Local with subscripts (array)	Passes the value of a single node	The standard way in which these variables are passed
Global variable with or without subscripts	Passes the value of a single node	Cannot be passed this way (data for a global is not in memory)
Object Reference (OREF)*	The standard way in which these variables are passed	Allowed

* If you have a variable representing an object, you refer to the object by means of an object reference (OREF). When you pass an OREF as an argument, you typically pass it by value. However, since an OREF is a pointer to the object, you are effectively passing the object by reference. Changing the value of a property of the object inside the method changes the actual object, not a copy of the object. Passing an OREF by reference is allowed and can be used if you want to change the OREF to point to a different object. This is not a common usage. See Objects for more information on objects and object references.

6.6 Passing a Variable Number of Arguments

Some units of code can accept a variable number of arguments. For example:

- The reference page for the **$CLASSMETHOD** function shows the formal argument list as follows:

  ```
  $CLASSMETHOD(classname, methodname, arg1, arg2, arg3, ... )
  ```

  In this case, *arg1*, *arg2*, and *arg3* are placeholders, and the three trailing periods indicate that this function can accept a variable number of arguments.

- %SQL.Statement shows the formal argument list for the **%Execute()** method as follows:

  ```
  method %Execute(%parm...) as %SQL.StatementResult
  ```

  Notice the three periods after the argument. This syntax indicates that the method accepts a variable number of arguments.

With a variable number of arguments like this, you can pass a set of arguments—of varying number—to another unit of code, which has a argument list that may not be known in advance. In all cases when a variable number of arguments are accepted, you should list these arguments in the order expected by the applicable downstream unit of code. Separate these arguments with commas as usual. (Or create and pass a multidimensional array, as described in the subsection.)

For example, the **$CLASSMETHOD** function enables you to invoke a class method, passing to it any arguments of that method. Suppose that `MyPkg.MyClass` has a method with the following signature:

Class Member

```
ClassMethod SampleMethod(arg1 as %Integer,arg2 as %String,arg3 as %String) as %String {
}
```

You could invoke this method as follows:

ObjectScript

```
set a=10
set b="abc"
set c=$username
set myvar=$CLASSMETHOD("MyPkg.MyClass","SampleMethod",a,b,c)
```

Similarly, as seen above, the **%Execute()** method accepts a variable number of arguments, which are all passed, in order, to the query being executed by the %SQL.Statement instance.

For information on defining a method this way, see Specifying a Variable Number of Arguments.

6.6.1 Variation: Using a Multidimensional Array

When a method accepts a variable number of arguments, you can create and pass a multidimensional array that contains the arguments. This is best explained via an example, using the previous example method:

ObjectScript

```
set myargs(1)=10
set myargs(2)="abc"
set myargs(3)=$username
set myargs=3
set myvar=$CLASSMETHOD("MyPkg.MyClass","SampleMethod",myargs...)
```

Notice that the top node of the multidimensional array indicates the number of array elements and the subscripts are integers starting with 1. Also notice the three periods after the name of the multidimensional array.

This technique provides a useful way to pass a variable number of arguments when using %SQL.Statement. For example:

```
  Set sql="SELECT * FROM Test.Test WHERE A=?"
  Set params($INCREMENT(params))="value 0"
  If condition1 {
     Set sql=sql_" AND B=?"
     Set params($INCREMENT(params))="value 1"
   }
   If condition2 {
       Set sql=sql_" AND C=?"
       Set params($INCREMENT(params))="value 2"
   }
   Set statementResult=##class(%SQL.Statement).%ExecDirect(,sql,params...)
```

7 ObjectScript Variables and Scope

A variable is the name of a location in which a value can be stored. Unlike many computer languages, ObjectScript does not require variables to be declared. A variable is created when it is assigned a value.

7.1 Kinds of Variables

Within ObjectScript, there are multiple kinds of variables, as follows:

- Local variables, which hold data in memory.

  Local variables can have public or private scope.

- Global variables or *globals*, which hold data in a database. All interactions with a global affect the database immediately. For example, when you set the value of a global, that change immediately affects what is stored; there is no separate step for storing values. Similarly, when you remove a global, the data is immediately removed from the database.
- Process-private global variables or PPGs
- i%property instance variables
- Special variables, also called *system variables*

This page primarily discusses local variables and global variables.

7.2 Variable Names

The name of a variable determines what kind of variable it is. The names of variables follow these rules:

- For most local variables, the first character is a letter, and the rest of the characters are letters or numbers. Valid names include `myvar` and `i`
- For most global variables, the first character is always a caret (`^`). The rest of the characters are letters, numbers, or periods. Valid names include `^myvar` and `^my.var`

  Note that InterSystems IRIS provides special treatment for globals with names that start `^IRIS.TempUser` — for example, `^IRIS.TempUser.MyApp`. If you create such globals, these globals are written to the IRISTEMP database.

- For a process-private global variable, the first character is a caret. Valid names include `^||MyVar`, `^|"^"|MyVar`, `^["^"]MyVar`, and `^["^",""]MyVar`. See Process-Private Globals.

For information on names that avoid collisions with system code, see Rules and Guidelines for Identifiers.

7.2.1 Percent Variables

For local and global variables, InterSystems IRIS supports a variation known as a *percent variable*; these are less common. The name of a local percent variable starts with `%`, and the name of a global percent variable starts with `^%`. Percent variables are special in that they are always public; that is they are visible to all code within a process. This includes all methods and all procedures within the calling stack. In the case of a global percent variable, in addition to being public, the variable is available in all namespaces.

To avoid collisions with system code, when you define percent variables, use the following rules:

- For a local percent variable, start the name with `%Z` or `%z`.
- For a global percent variable, start the name with `^%Z` or `^%z`.

For further details on variable names and for variations, see Rules and Guidelines for Identifiers.

7.3 Variable Availability and Scope

ObjectScript supports the following program flow, which is similar (in most ways) to what other programming languages support:

1. A user invokes a method, perhaps from a user interface.
2. The method executes some statements and then invokes a second method.
3. The second method defines local variables A, B, and C.

   Variables A, B, and C are *in scope* within this method. They are *private* to this method.
4. The second method also defines the global variable ^D.
5. The second method ends, and control returns to the first method.
6. The first method resumes execution. This method cannot use variables A, B, and C, which are no longer defined. It can use ^D, because that variable was immediately saved to the database.

The preceding program flow is quite common. InterSystems IRIS provides other options, however, of which you should be aware.

7.3.1 Default Variable Scope

Several factors control whether a variable is available outside of the method that defines it. Before discussing those, it is necessary to point out the following environmental details:

- An InterSystems IRIS instance includes multiple namespaces, including multiple system namespaces and probably multiple namespaces that you define.
- You can run multiple processes simultaneously in a namespace. In a typical application, many processes are running at the same time.

The following table summarizes where variables are available (except for process-private global variables, described on another page):

Variable availability, broken out by kind of variable	Outside of code that defines it (but in the same process)	In other processes in the same namespace	In other namespaces within same InterSystems IRIS instance
Local variable, private scope*	No	No	No
Local variable, public scope	Yes	No	No
Local percent variable	Yes	No	No
Global variable (not percent)	Yes	Yes	Not unless global mappings permit this†
Global percent variable	Yes	Yes	Yes

*By default, variables defined in a procedure or a method are private to that procedure or method, as noted before. Also, in a procedure or method, you can declare variables as public variables, although this practice is not preferred. See PublicList.

†Each namespace has default databases for specific purposes and can have mappings that give access to additional databases. Consequently, a global variable can be available to multiple namespaces, even if it is not a global percent variable. See Namespaces and Databases.

7.3.2 The NEW Command

InterSystems IRIS provides another mechanism to enable you to control the scope of a variable: the **NEW** command. The argument to this command is one or more variable names, in a comma-separated list. The variables must be public variables and cannot be global variables.

This command establishes a new, limited context for the variable (which may or may not already exist). For example, consider the following routine:

ObjectScript

```
 ; demonew
 ; routine to demo NEW
 NEW var2
 set var1="abc"
 set var2="def"
 quit
```

After you run this routine, the variable `var1` is available, and the variable `var2` is not, as shown in the following example Terminal session:

```
TESTNAMESPACE>do ^demonew

TESTNAMESPACE>write var1
abc
TESTNAMESPACE>write var2

write var2
^
<UNDEFINED> *var2
```

If the variable existed before you used **NEW**, the variable still exists after the scope of **NEW** has ended, and it retains its previous value. For example, consider the following Terminal session, which uses the routine defined previously:

```
TESTNAMESPACE>set var2="hello world"

TESTNAMESPACE>do ^demonew

TESTNAMESPACE>write var2
hello world
```

7.4 Variable Length

The length of a value of a variable must be less than the string length limit.

7.5 Variable Existence and Undefined Variables

You usually define a variable with the **SET** command. As noted earlier, when you define a global variable, that immediately affects the database.

A global variable becomes undefined only when you *kill* it (which means to remove it via the **KILL** command). This also immediately affects the database.

A local variable can become undefined in one of three ways:

- It is killed.
- The process (in which it was defined) ends.
- It goes out of scope within that process.

To determine whether a variable is defined, you use the **$DATA** function. For example, the following shows a Terminal session that uses this function:

```
TESTNAMESPACE>write $DATA(x)
0
TESTNAMESPACE>set x=5

TESTNAMESPACE>write $DATA(x)
1
```

In the first step, we use **$DATA** to see if a variable is defined. The system displays 0, which means that the variable is not defined. Then we set the variable equal to 5 and try again. Now the function returns 1.

If you attempt to access an undefined variable, you get the <UNDEFINED> error. For example:

```
TESTNAMESPACE>WRITE testvar

WRITE testvar
^
<UNDEFINED> *testvar
```

7.6 #dim (Optional)

ObjectScript does not require variables to be declared. You can, however, use the #dim preprocessor directive as an aid to documenting and writing code; with this directive, IDEs can provide type hints.

The syntax forms of **#dim** are:

ObjectScript

```
#dim VariableName As DataTypeName
#dim VariableName As List Of DataTypeName
#dim VariableName As Array Of DataTypeName
```

where *VariableName* is the variable for which you are naming a data type and *DataTypeName* specifies that data type.

7.7 Global Variables and Journaling

InterSystems IRIS treats a **SET** or **KILL** of a global as a journaled transaction event; rolling back the transaction reverses these operations. Locks may be used to prevent access by other processes until the transaction that made the changes has been committed. Refer to Transaction Processing for further details.

In contrast, InterSystems IRIS does not treat a **SET** or **KILL** of a local variable or a process-private global as a journaled transaction event; rolling back the transaction has no effect on these operations.

7.8 See Also

- Multidimensional Arrays
- Working with Globals
- Process-Private Globals

8

Process-Private Globals

An ObjectScript *process-private global* is a variable that is only accessible by the process that created it. When the process ends, all of its process-private globals are deleted.

Process-private globals are written to the IRISTEMP database. In contrast to global variables, InterSystems IRIS does not treat a SET or KILL of a local variable or a process-private global as a journaled transaction event; rolling back the transaction has no effect on these operations.

8.1 Introduction

A process-private global has the following characteristics:

- Process-specific: a process-private global can only be accessed by the process that created it, and it ceases to exist when the process completes. This is similar to local variables.
- Always public: a process-private global is always a public variable. This is similar to global variables.
- Namespace-independent: a process-private global is accessible from all namespaces.
- Unaffected by argumentless **KILL**, **NEW**, **WRITE**, or **ZWRITE**. A process-private global can be specified as an argument to **KILL**, **WRITE**, or **ZWRITE**. This is similar to global variables.

8.2 Naming Conventions

A process-private global name takes one of the following forms:

```
^||name
^|"^"|name
^["^"]name
^["^",""]name
```

These four prefix forms are equivalent, and all four refer to the same process-private global. The first form (^||name) is the most common, and the one recommended for new code. The second, third, and fourth forms are provided for compatibility with existing code that defines globals.

Apart from the prefix, process-private globals use the same naming conventions as regular globals, as given in Rules and Guidelines for Identifiers. Briefly:

- The first character (after the second vertical bar) must be either a letter or the percent (%) character.

Process-private variable names starting with `%` are known as “percent variables” and have different scoping rules. In your code, for these variables, start the name with `%Z` or `%z`; other names are reserved for system use. For example: `^||%zmyvar`.

- Unlike local variables, no global name (including process-private globals) can contain Unicode letters — letter characters above ASCII 255. Attempting to include a Unicode letter in a process-private global name results in a <WIDE CHAR> error.
- All variable names are case-sensitive, and this includes process-private global names.
- A process-private global name must be unique within its process.
- Unlike local variables, process-private global names are limited to 31 characters, exclusive of the prefix characters. You may specify a name longer than 31 characters, but only the first 31 characters are used. Therefore, a process-private global name must be unique within its first 31 characters.
- Like other variables, process-private globals can take subscripts.

8.3 Listing Process-Private Globals

You can use the **^$||GLOBAL()** syntax form of ^$GLOBAL() to return information about process-private globals belonging to the current process.

You can use the **^GETPPGINFO** routine to display the names of all current process-private globals and their space allocation, in blocks. **^GETPPGINFO** does not list the subscripts or values for process-private globals. You can display process-private globals for a specific process by specifying its process Id (pid), or for all processes by specifying the "*" wildcard string. You must be in the %SYS namespace to invoke **^GETPPGINFO**.

The following example uses **^GETPPGINFO** to list the process-private globals for all current processes:

ObjectScript

```
  SET ^||flintstones(1)="Fred"
  SET ^||flintstones(2)="Wilma"
  NEW $NAMESPACE
  SET $NAMESPACE="%SYS"
  DO ^GETPPGINFO("*")
```

The **^GETPPGINFO** routine takes arguments as follows:

ObjectScript

```
 do ^GETPPGINFO("pdf","options","outfile")
```

These arguments are as follows:

- *pdf* can be a process Id or the * wildcard.
- *options* can be a string containing any combination of the following characters:
  - `b` (return values in bytes)
  - `M`*nn* (list only processes with process-private globals that use *nn* or more blocks)

    Use `M0` to include processes without any process-private globals in the listing.

    Use `M1` to exclude processes without any process-private globals from the listing, but include processes having only a global directory block. (This is the default.)

  Use `M2` to exclude processes without any process-private globals from the listing, as well as those having only a global directory block.

  - `S` (suppress screen display; used with *outfile*)
  - `T` (display process totals only).

- *outfile* is the file path for a file in CSV (comma-separated values) format that will be used to receive **^GETPPGINFO** output.

The following example writes process-private globals to an output file named `ppgout`. The S option suppresses screen display; the M500 option limits output to only processes with process-private globals that use 500 or more blocks:

ObjectScript

```
  NEW $NAMESPACE
  SET $NAMESPACE="%SYS"
  DO ^GETPPGINFO("*","SM500","/home/myspace/ppgout")
```

You can also query the %SYS.ProcessQuery table for information on process-private globals. For example:

SQL

```
SELECT PID, Routine, PrivateGlobalBlockCount
       FROM %SYS.ProcessQuery
       WHERE PrivateGlobalBlockCount>0
       ORDER BY PrivateGlobalBlockCount DESC
```

8.4 See Also

- Variables in ObjectScript
- Multidimensional Arrays

9

Extended References

In InterSystems IRIS® data platform, all code that runs on the server runs in a namespace; see Namespaces and Databases. ObjectScript supports *extended references*, which is syntax that enables you to refer to specific kinds of items in other namespaces.

9.1 Introduction

An extended reference is a reference to an entity that is located in another namespace. The namespace name can be specified as a string literal enclosed in quotes, as a variable that resolves to a namespace name, as an implied namespace name, or as a null string ("") a placeholder that specifies the current namespace.

All extended references can specify the *current* namespace, either explicitly by name, or by specifying a null string placeholder.

9.2 Types

There are three types of extended references:

- An *extended global reference* references a global variable in another namespace. The following syntactic forms are supported:

  ```
  ^["namespace"]global
  ^|"namespace"|global
  ```

  For further details, see Extended Global References.

- An *extended routine reference* references a routine in another namespace.
  - The DO command, the $TEXT function, and user-defined functions support the following syntactic form:

    ```
    |"namespace"|routine
    ```

  - The JOB command supports the following syntactic forms: :

    ```
    routine|"namespace"|
    routine["namespace"]
    routine:"namespace"
    ```

In all these cases, the reference is prefaced by a ^ (caret) character to indicate that the specified entity is a routine (rather than a label or an offset). This caret is not part of the routine name. For example, `DO ^|"SAMPLES"|fibonacci` invokes the routine named fibonacci, which is located in the SAMPLES namespace. The command `WRITE $$fun^|"SAMPLES"|house` invokes the user-defined function fun() in the routine house, located in the SAMPLES namespace.

- An *extended SSVN reference* references a structured system variable (SSVN) in another namespace. The following syntactic forms are supported:

  ```
  ^$["namespace"]ssvn
  ^$|"namespace"|ssvn
  ```

  For further details, refer to the ^$GLOBAL, ^$LOCK, and ^$ROUTINE structured system variables.

9.3 See Also

- Extended Global References

10 Multidimensional Arrays

ObjectScript includes support for multidimensional arrays, which you can use to contain and manipulate values that are related in some manner. Multidimensional arrays are supported extensively in ObjectScript and in InterSystems IRIS® data platform.

There are class-based alternatives as well.

10.1 Introduction and Terminology

A multidimensional array is a structure consisting of one or more nodes, and each node is identified by a unique subscript or set of subscripts. For example, the *MyVar* array could consist of the following nodes:

- MyVar
- MyVar(22)
- MyVar(-3)
- MyVar(“MyString”)
- MyVar(-123409, “MyString”)
- MyVar(“MyString”, 2398)
- MyVar(1.2, 3, 4, “Five”, “Six”, 7)

Each node of an array is an ObjectScript variable and holds a single value of any type; you can work with any node in exactly the same way that you work with a variable that does not have a subscript. ObjectScript provides specialized functions that work with the array itself, enabling you to traverse it, selectively remove parts of it, and perform other actions applicable to the structure as a whole.

10.1.1 Multidimensional Tree Structures

The entire structure of a multidimensional array is called a *tree*; it begins at the top and grows downwards. The *root*, *MyVar* above, is at the top. The root, and any other subscripted form of it, are called *nodes*. Nodes that have no nodes beneath them are called *leaves*. Nodes that have nodes beneath them are called *parents* or *ancestors*. Nodes that have parents are called *children* or *descendants*. Children with the same parents are called *siblings*. All siblings are automatically sorted numerically or alphabetically as they are added to the tree.

10.1.2 Sparse Multidimensional Storage

Multidimensional arrays are sparse. This means that the example above uses only seven reserved memory locations, one for each defined node. Further, since there is no need to declare arrays or specify their dimensions, there are additional memory benefits: no space is reserved for them ahead of time; they use no space until needing it; and all the space that they use is dynamically allocated. As an example, consider an array used to keep track of players' pieces for a game of checkers; a checkerboard is 8 by 8. In a language that required an 8–by-8 checkerboard-sized array would use 64 memory locations, even though no more than 24 positions are ever occupied by checkers; in ObjectScript, the array would require 24 positions only at the beginning, and would need fewer and fewer during the course of the game.

10.2 Where Multidimensional Arrays Are Supported

Local variables, process-private variables, and global variables can all be multidimensional arrays. Lock names can also be multidimensional arrays. It is not necessary to do any kind of declaration of these items as multidimensional arrays.

Also, a property in a class can be a multidimensional array if it has the `MultiDimensional` keyword in its definition, for example:

```
Property MyProp as %String [ MultiDimensional ];
```

The purpose of this declaration (the `MultiDimensional` keyword) is to affect the code generated for the class. This kind of property cannot be saved to the database.

The phrase “to support an item as a multidimensional array” means that it is syntactically valid to set or refer to a subscript of the item. An alternate phrasing of the idea is “to allow an item to have subscripts.”

For example, *MyVar* is a local variable. Because local variables are supported as multidimensional arrays, it is syntactically valid to use commands like these:

```
 Set MyVar("test subscript")=45
 Write MyVar("test subscript")
```

As described in Subscript Rules, you can use multiple subscripts, not just the single one shown here.

10.3 Subscript Rules

The following list describes the rules for subscripts of multidimensional arrays:

- A subscript can be a numeric or a string. It can include any characters, including Unicode characters. Valid numeric subscripts include positive and negative numbers, zero, and fractional numbers.
- The empty string (`""`) is *not* a valid subscript.
- Subscript values are case-sensitive.
- Any numeric subscript is converted to canonical form. Thus, for example, the global nodes `^a(7)`, `^a(007)`, `^a(7.000)`, and `^a(7.)` are all the same because the subscript is actually the same in all cases.
- A string subscript is *not* converted to canonical form. Thus, for example, `^a("7")`, `^a("007")`, `^a("7.000")`, and `^a("7.")` are all different global nodes because these subscripts are all different. Also, `^a("7")` and `^a(7)` both refer to the same global node, because these subscripts are the same.

- An array node can have multiple subscripts, and the subscripts are not required to have the same data type or to follow any specific system.
- There are limits on the length of a subscript and on the number of subscript levels. See Subscript Limits.

Also see General System Limits.

10.4 Manipulating Multidimensional Arrays

InterSystems IRIS provides a comprehensive set of commands and functions for working with multidimensional arrays:

- Set places values in an array.
- Kill removes all or part of an array structure.
- Merge copies all or part of an array structure to a second array structure.
- $Order and $Query allows you to iterate over the contents of an array.
- $Data allows you to test for the existence of nodes in an array.

For details on these commands, see Working with Globals, which applies to globals and all other kinds of multidimensional arrays.

Also, you can use the $QSUBSCRIPT function to return the components (name and subscripts) of a specified variable, or the $QLENGTH function to return the number of subscript levels.

10.5 Class-Based Arrays

Rather than multidimensional arrays, you can use array classes that use a different structure and that provide an object-based API. InterSystems IRIS provides a set of classes you can use as array-form class properties, and another set of classes you can use for standalone arrays.

10.6 See Also

- General System Limits
- Working with Globals
- Working with Collection Classes

11 Types of Data in ObjectScript

This page discusses types of data supported in ObjectScript.

Note: Although ObjectScript is typeless, InterSystems IRIS® data platform does support and enforce types for class properties and table fields. There is an extensive set of data type classes, and InterSystems SQL supports standard SQL data types.

11.1 Introduction

Formally, ObjectScript is a typeless language — you do not have to declare the types of variables. Any variable can contain any kind of value, and usage determines how the value is evaluated. For example, `5` can be treated as a number, a string, or a boolean value. Phrased differently, variables in ObjectScript are weakly, dynamically typed. They are dynamically typed because you do not have to declare the type for a variable, and variables can take any legal value. They are weakly typed because usage determines how they are evaluated.

11.2 Common Types

Although ObjectScript variables do not have types, it is possible to categorize the types of values they can contain, in a generic sense. The most commonly used types of values are as follows:

numbers

A number is a set of digits, including a leading plus or minus sign, a decimal sign, and an exponentiation sign, if needed.

ObjectScript supports two internal representations of fractional numbers: standard InterSystems IRIS floating point numbers ($DECIMAL numbers) and IEEE double-precision floating-point numbers ($DOUBLE numbers).

See Numeric Values in ObjectScript.

strings

A string is a set of characters: letters, digits, punctuation, and so on delimited by a matched set of quotation marks ("):

ObjectScript

```
SET string = "This is a string"
WRITE string
```

You can include a " (double quote) character as a literal within a string by preceding it with another double quote character:

ObjectScript

```
SET string = "This string has ""quotes"" in it."
WRITE string
```

See Strings in ObjectScript. Also see String Length Limit.

lists

ObjectScript provides a native list format. See Lists in ObjectScript, which also discusses alternatives.

date and date/time values

ObjectScript has no built-in date type; instead, it includes a number of functions for operating on and formatting date values represented as strings. See Date and Time Values in ObjectScript.

boolean values

Specific ObjectScript operators, functions, and commands can treat any value as a boolean value (true or false).

Unlike some other languages, ObjectScript does not provide specialized representations of boolean literal values. For simplicity, use 1 for true, and 0 for false.

See Boolean Values in ObjectScript.

OREFs

A variable can contain an OREF, which is a handle to an in-memory object. An OREF is also called an object value. You can assign an OREF to any local variable:

ObjectScript

```
 SET myperson = ##class(Sample.Person).%New()
 WRITE myperson
```

A global variable cannot equal an OREF. A runtime error occurs if you try to make such an assignment.

See Working with Registered Objects and Working with Persistent Objects.

11.3 Type Conversion Summary

Depending on the context, a string can be treated as a number and vice versa. Similarly, in some contexts, a value may be interpreted as a boolean (true or false) value; anything that evaluates to zero is treated as false; anything else is treated as true. This means that you can assign a string value to a variable and, later on, assign a numeric value to the same variable. As an optimization, InterSystems IRIS may use different internal representations for strings, integers, numbers, and objects, but this is not visible to the application programmer. InterSystems IRIS automatically converts (or interprets) the value of a variable based on the context in which it is used.

The following table summarizes how InterSystems IRIS converts values:

Table 11–1: ObjectScript Type Conversion Rules

From	To	Rules
Number	String	A string of characters that represents the numeric value is used, such as `2.2` for the variable *num* in the previous example.
String	Number	Leading characters of the string are interpreted as a numeric literal, as described in String-to-Number Conversion. For example, “–1.20abc” is interpreted as `-1.2` and “abc123” is interpreted as `0`.
Object	Number	The internal object instance number of the given object reference is used. The value is an integer.
Object	String	A string of the form *n@cls* is used, where *n* is the internal object instance number and *cls* is the class name of the given object.
Number	Object	Not allowed.
String	Object	Not allowed.

12
Numeric Values in ObjectScript

This page describes numeric values in ObjectScript, as well as operators for working with them.

12.1 Numeric Literals

Numeric literals do not require any enclosing punctuation. You can specify a number using any valid numeric characters. InterSystems IRIS evaluates a number as syntactically valid, then converts it to canonical form.

The syntactic requirements for a numeric literal are as follows:

- It can contain the decimal numbers 0 through 9, and must contain at least one of these number characters. It can contain leading or trailing zeros. However, when InterSystems IRIS converts a number to canonical form it automatically removes leading integer zeros. Therefore, numbers for which leading integer zeros are significant must be input as strings. For example, United State postal Zip Codes can have a leading integer zero, such as 02142, and therefore must be handled as strings, not numbers.
- It can contain any number of leading plus and minus signs in any sequence. However, a plus sign or minus sign cannot appear after any other character, except the "E" scientific notation character. In a numeric expression a sign after a non-sign character is evaluated as an addition or subtraction operation. In a numeric string a sign after a non-sign character is evaluated as a non-numeric character, terminating the number portion of the string.

  InterSystems IRIS uses the PlusSign and MinusSign property values for the current locale to determine these sign characters ("+" and "-" by default); these sign characters are locale-dependent. To determine the PlusSign and MinusSign characters for your locale, invoke the **GetFormatItem()** method:

  ObjectScript

  ```
  WRITE ##class(%SYS.NLS.Format).GetFormatItem("PlusSign"),!
  WRITE ##class(%SYS.NLS.Format).GetFormatItem("MinusSign")
  ```

- It can contain at most one decimal separator character. In a numeric expression a second decimal separator results in a <SYNTAX> error. In a numeric string a second decimal separator is evaluated as the first non-numeric character, terminating the number portion of the string. The decimal separator character may be the first character or the last character of the numeric expression. The choice of decimal separator character is locale-dependent: American format uses a period (.) as the decimal separator, which is the default. European format uses a comma (,) as the decimal separator. To determine the DecimalSeparator character for your locale, invoke the **GetFormatItem()** method:

  ObjectScript

  ```
  WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
  ```

- It can contain at most one letter "E" (or "e") to specify a base-10 exponent for scientific notation. This scientific notation character ("E" or "e") must be preceded by a integer or fractional number, and followed by an integer.

Numeric literal values *do not* support the following:

- They cannot contain numeric group separators. These are locale-dependent: American format uses commas, European format uses periods. You can use the **$INUMBER** function to remove numeric group separators, and the **$FNUMBER** function to add numeric group separators.
- They cannot contain currency symbols, hexadecimal letters, or other nonnumeric characters. They cannot contain blank spaces, except before or after arithmetic operators.
- They cannot contain trailing plus or minus signs. However, the **$FNUMBER** function can display a number as a string with a trailing sign, and the **$NUMBER** function can take a string in this format and convert it to a number with a leading sign.
- They cannot specify enclosing parentheses to represent a number as a negative number (a debit). However, the **$FNUMBER** function can display a negative number as a string with a enclosing parentheses, and the **$NUMBER** function can take a string in this format and convert it to a number with a leading negative sign.

A number or numeric expression can containing pairs of enclosing parentheses. These parentheses are not part of the number, but govern the precedence of operations. By default, InterSystems IRIS performs all operations in strict left-to-right order.

12.1.1 Scientific Notation

To specify scientific (exponential) notation in ObjectScript, use the following format:

`[-]`*`mantissa`*`E[-]`*`exponent`*

where

Element	Description
-	*Optional* — One or more Unary Minus or Unary Plus operators. These PlusSign and MinusSign characters are configurable. Conversion to canonical form resolves these operators after resolving the scientific notation.
mantissa	An integer or fractional number. May contain leading and trailing zeros and a trailing decimal separator character.
E	An operator delimiting the exponent. The uppercase "E" is the standard exponent operator; the lowercase "e" is a configurable exponent operator, using the **ScientificNotation()** method of the %SYSTEM.Process class.
-	*Optional* — A single Unary Minus or Unary Plus operator. Can be used to specify a negative exponent. These PlusSign and MinusSign characters are configurable.
exponent	An integer specifying the exponent (the power of 10). Can contain leading zeros. Cannot contain a decimal separator character.

For example, to represent 10, use `1E1`. To represent 2800, use `2.8E3`. To represent .05, use `5E-2`.

No spaces are permitted between the *mantissa*, the `E`, and the *exponent*. Parentheses, concatenation, and other operators are not permitted within this syntax.

Because resolving scientific notation is the first step in converting a number to canonical form, some conversion operations are not available. The *mantissa* and *exponent* must be numeric literals, they cannot be variables or arithmetic expressions. The *exponent* must be an integer with (at most) one plus or minus sign.

See the **ScientificNotation()** method of the %SYSTEM.Process class.

12.2 Arithmetic

The arithmetic operators interpret their operands as numeric values and produce numeric results. When operating on a string, an arithmetic operator treats the string as its numeric value, according to the rules in String-to-Number Conversion. The arithmetic operators are as follows:

Unary Positive (+)

The unary positive operator gives its single operand a numeric interpretation. It does this by sequentially parsing the characters of the string as a number, until it encounters a character that cannot be interpreted as a number. It then returns whatever leading portion of the string was a well-formed numeric (or it returns 0 if no such interpretation was possible). For example:

Terminal

```
USER>WRITE + "32 dollars and 64 cents"
32
```

Unary Negative (-)

The unary negative operator reverses the sign of a numerically interpreted operand. For example:

Terminal

```
USER>SET x = -60

USER>WRITE x
-60
USER>WRITE -x
60
```

ObjectScript gives the unary negative operator precedence over the binary (two-operand) arithmetic operators.

To return the absolute value of a numeric expression, use the **$ZABS** function.

Addition (+)

The addition operator adds two numeric values. For example:

Terminal

```
USER>WRITE 2936.22 + 301.45
3237.67
```

Subtraction (-)

The subtraction operator subtracts one numeric value from another. For example:

Terminal

```
USER>WRITE 2936.22 - 301.45
2634.77
```

Multiplication (*)

The multiplication operator multiplies two numeric values. For example:

Terminal

```
USER>WRITE 9 * 5.5
49.5
```

Division (/)

The division operator divides one numeric value with another. For example:

Terminal

```
USER>WRITE 355 / 113
3.141592920353982301
```

Integer Division (\ )

The integer division operator divides one numeric value with another and discards any fractional value. For example:

Terminal

```
USER>WRITE 355 \ 113
3
```

Modulo (#)

When the two operands are positive, then the modulo operator returns the remainder of the left operand integer divided by the right operand. For example:

Terminal

```
USER>WRITE 37 # 10
7
USER>WRITE 12.5 # 3.2
2.9
```

Exponentiation (**)

The exponentiation operator raises one numeric value to the power of the other numeric value. For example:

Terminal

```
USER>WRITE 9 ** 2
81
```

Exponentiation can also be performed using the **$ZPOWER** function.

12.3 Numeric Equality

You can use the equals (=) and does not equal ('=) operators to compare two numbers.

The equals operator tests whether the left operand equals the right operand. For example:

Terminal

```
USER>WRITE 9 = 6
0
```

You can also use the does not equal ('=) operator.

12.4 Other Numeric Comparisons

The equals (=) and does not equal ('=) operators, discussed previously, can be used with strings. In contrast, the operators discussed here are meant for use only with numeric values. The operators here *always* interpret their operands as numeric values (see String-to-Number Conversion) and then produce a Boolean result.

Less Than Operator (<)

The less than operator tests whether the left operand is less than the right operand. For example:

Terminal

```
USER>WRITE 9 < 6
0
```

Greater Than Operator (>)

The greater than operator tests whether the left operand is greater than the right operand. For example:

Terminal

```
USER>WRITE 15 > 15
0
```

Less Than or Equal To Operator (<= or '>)

The less than or equal to operator tests whether the left operand is less or equal to than the right operand. For example:

Terminal

```
USER>9 <= 6
0
```

Greater Than or Equal To Operator (>= or '<)

The greater than or equal to operator tests whether the left operand is greater than or equal to the right operand. For example:

Terminal

```
USER>WRITE 15 >= 15
1
```

12.5 Canonical Numbers

ObjectScript performs all numeric operations on numbers in their canonical form. For example, the length of the number `+007.00` is 1; the length of the string `"+007.00"` is 7.

When InterSystems IRIS converts a number to canonical form, it performs the following steps:

1. Scientific notation exponents are resolved. For example 3E4 converts to 30000 and 3E-4 converts to .0003.

2. Leading signs are resolved. First, multiple signs are resolved to a single sign (for example, two minus signs resolve to a plus sign). Then, if the leading sign is a plus sign, it is removed. You can use the **$FNUMBER** function to explicitly specify (prepend) a plus sign to a positive InterSystems IRIS canonical number.

   Note: ObjectScript resolves any combination of leading plus and minus signs. In SQL, two consecutive minus signs are parsed as a single-line comment indicator. Therefore, specifying a number in SQL with two consecutive leading minus signs results in an SQLCODE -12 error.

3. All leading and trailing zeros are removed. This includes removing leading integer zeroes, including the leading integer zero from fractions smaller than 1. For example `0.66` becomes `.66`.

   - To append an integer zero to a canonical fraction use the **$FNUMBER** or **$JUSTIFY** function. `.66` becomes `0.66`.
   - To remove integer zeroes from a non-canonical fraction use the Unary Plus operator to force conversion of a number string to a canonical number. In the following example, the fractional seconds portion of a timestamp, `+$PIECE("65798,00000.66",",",2)`. `00000.66` becomes `.66`.

   As part of this conversion, zero fractions are simplified to 0. Regardless of how expressed (`0.0`, `.0`, `.000`) all zero values are converted to `0`.

4. A trailing decimal separator is removed.
5. -0 is converted to 0.
6. Arithmetic operations and numeric concatenation are performed. InterSystems IRIS performs these operations in strict left-to-right order. Numbers are in their canonical form when these operations are performed. For further details, see Concatenating Numbers below.

InterSystems IRIS canonical form numbers differ from other canonical number formats used in InterSystems software:

- ODBC: Integer zero fractions converted to ODBC have a zero integer. Therefore, `.66` and `000.66` both become `0.66`. You can use the $FNUMBER or $JUSTIFY function to prepend an integer zero to an InterSystems IRIS canonical fractional number.
- JSON: Only a single leading minus sign is permitted; a leading plus sign or multiple signs are not permitted.

  Exponents are permitted but not resolved. 3E4 is returned as 3E4.

  Leading zeros are not permitted. Trailing zeros are not removed.

  Integer zero fractions must have a zero integer. Therefore, `.66` and `000.66` are not valid JSON numbers, but `0.66` and `0.660000` are valid JSON numbers.

  A trailing decimal separator is not permitted.

  Zero values are not converted: `0.0`, `-0`, and `-0.000` are returned unchanged as valid JSON numbers.

12.5.1 Concatenating Numbers

A number can be concatenated to another number using the concatenate operator (_). InterSystems IRIS first converts each number to its canonical form, then performs a string concatenation on the results. Thus, the following all result in 1234: 12_34, 12_+34, 12_--34, 12.0_34, 12_0034.0, 12E0_34. The concatenation 12._34 results in 1234, but the concatenation 12_.34 results in 12.34. The concatenation 12_-34 results in the string “12-34”.

InterSystems IRIS performs numeric concatenation and arithmetic operations on numbers after converting those numbers to canonical form. It performs these operations in strict left-to-right order, unless you specify parentheses to prioritize an operation. The following example explains one consequence of this:

ObjectScript

```
WRITE 7_-6+5 // returns 12
```

In this example, the concatenation returns the string “7-6”. This, of course, is not a canonical number. InterSystems IRIS converts this string to a canonical number by truncating at the first non-numeric character (the embedded minus sign). It then performs the next operation using this canonical number 7 + 5 = 12.

12.6 Floating-Point Numbers

InterSystems IRIS supports two different numeric types that can be used to represent floating-point numbers:

- Decimal floating-point: By default, InterSystems IRIS represents fractional numbers using its own decimal floating-point standard ($DECIMAL numbers). This is the preferred format for most uses. It provides a higher level of precision than IEEE Binary floating-point. It is consistent across all system platforms that InterSystems IRIS supports. Decimal floating-point is preferred for data base values. In particular, a fractional number such as 0.1 can be exactly represented using decimal floating-point notation, while the fractional number 0.1 (as well as most decimal fractional numbers) can only be approximated by IEEE Binary floating-point.

  Internally, Decimal arithmetic is performed using numbers of the form M*(10**N), where M is the integer significand containing an integer value between -9223372036854775808 and 9223372036854775807 and N is the decimal exponent containing an integer value between -128 and 127. The significand is represented by a 64-bit signed integer and the exponent is represented by an 8-bit signed byte.

  The average precision of Decimal floating point is 18.96 decimal digits. Decimal numbers with a significand between 1000000000000000000 and 9223372036854775807 have exactly 19 digits of precision and a Decimal significant between 922337203685477581 and 999999999999999999 have exactly 18 digits of precision. Although IEEE Binary floating-point is less precise (with an accuracy of approximately 15.95 decimal digits), the exact, infinitely precise value of IEEE Binary representation as a decimal string can have over 1000 significant decimal digits.

  In the following example, **$DECIMAL** functions take a fractional number and an integer with 25 digits and return a Decimal number rounded to 19 digits of precision / 19 significant digits:

  #### Terminal

  ```
  USER>WRITE $DECIMAL(1234567890.123456781818181)
  1234567890.123456782
  USER>WRITE $DECIMAL(1234567890123456781818181)
  1234567890123456782000000
  ```

- IEEE Binary floating-point: IEEE double-precision binary floating point is an industry-standard way of representing fractional numbers. IEEE floating point numbers are encoded using binary notation. Binary floating-point representation is usually preferred when doing high-speed calculations because most computers include high-speed hardware for binary floating-point arithmetic.

  Internally, IEEE Binary arithmetic is performed using numbers of the form S*M*(2**N), where S is the sign containing the value -1 or +1, M is the significand containing a 53-bit binary fractional value with the binary point between the first and second binary bit, and N is the binary exponent containing an integer value between -1022 and 1023. Therefore, the representation consists of 64 bits, where S is a single sign bit, the exponent N is stored in the next 11 bits (with two additional values reserved), and the significand M is >=1.0 and <2.0 containing the last 52 bits with a total of 53 binary bits of precision. (Note that the first bit of M is always a 1, so it does not need to appear in the 64-bit representation.)

  Double-precision binary floating point has a precision of 53 binary bits, which corresponds to approximately 15.95 decimal digits of precision. (The corresponding decimal precision varies between 15.35 and 16.55 digits.)

Binary representation does not correspond exactly to a decimal fraction because a fraction such as 0.1 cannot be represented as a finite sequence of binary fractions. Because most decimal fractions cannot be exactly represented in this binary notation, an IEEE floating point number may differ slightly from the corresponding InterSystems Decimal floating point number. When an IEEE floating point number is displayed as a fractional number, the binary bits are often converted to a fractional number with far more than 18 decimal digits. This *does not* mean that IEEE floating point numbers are more precise than InterSystems Decimal floating point numbers. IEEE floating point numbers are able to represent larger and smaller numbers than InterSystems Decimal numbers.

In the following example, the **$DOUBLE** function take a sequence of 17-digit integers and returns values with roughly 16 significant digits of decimal precision:

Terminal

```
USER>FOR i=12345678901234558:1:12345678901234569 {W $DOUBLE(i),!}
12345678901234558
12345678901234560
12345678901234560
12345678901234560
12345678901234562
12345678901234564
12345678901234564
12345678901234564
12345678901234566
12345678901234568
12345678901234568
12345678901234568
```

IEEE Binary floating-point supports the special values INF (infinity) and NAN (not a number). For further details, see the $DOUBLE function.

You can configure processing of IEEE floating point numbers using the `IEEEError` setting for handling of INF and NAN values, and the `ListFormat` setting for handling compression of IEEE floating point numbers in $LIST structured data. Both can be viewed and set for the current process using %SYSTEM.Process class methods (**$SYSTEM.Process.IEEEError()**). System-wide defaults can be set using the InterSystems IRIS Management Portal, as follows: from **System Administration**, select **Configuration**, **Additional Settings**, **Compatibility**.

You can use the $DOUBLE function to convert an InterSystems IRIS standard floating-point number to an IEEE floating point number. You can use the $DECIMAL function to convert an IEEE floating point number to an InterSystems IRIS standard floating-point number.

By default, InterSystems IRIS converts fractional numbers to canonical form, eliminating all leading zeros. Therefore, `0.66` becomes `.66`. $FNUMBER (most formats) and $JUSTIFY (3-parameter format) always return a fractional number with at least one integer digit; using either of these functions, `.66` becomes `0.66`.

$FNUMBER and $JUSTIFY can be used to round or pad a numeric to a specified number of fractional digits. InterSystems IRIS rounds up 5 or more, rounds down 4 or less. Padding adds zeroes as fractional digits as needed. The decimal separator character is removed when rounding a fractional number to an integer. The decimal separator character is added when zero-padding an integer to a fractional number.

12.7 Extremely Large Numbers

The largest integers that can be represented exactly are the 19-digit integers -9223372036854775808 and 9223372036854775807. This is because these are the largest numbers that can be represented with 64 signed bits. Integers larger than this are automatically rounded to fit within this 64-bit limit. This is shown in the following example:

ObjectScript

```
SET x=9223372036854775807
WRITE x,!
SET y=x+1
WRITE y
```

Similarly, exponents larger that 128 may also result in rounding to permit representation within 64 signed bits. This is shown in the following example:

ObjectScript

```
WRITE 9223372036854775807e-128,!
WRITE 9223372036854775807e-129
```

Because of this rounding, arithmetic operations that result in numbers larger than these 19-digit integers have their low-order digits replaced by zeros. This can result in situations such as the following:

ObjectScript

```
SET longnum=9223372036854775790
WRITE longnum,!
SET add17=longnum+17
SET add21=longnum+21
SET add24=longnum+24
WRITE add17,!,add24,!,add21,!
IF add24=add21 {WRITE "adding 21 same as adding 24"}
```

The largest InterSystems IRIS decimal floating point number supported is 9.223372036854775807E145. The largest supported $DOUBLE value (assuming IEEE overflow to INFINITY is disabled) is 1.7976931348623157081E308. The $DOUBLE type supports a larger range of values than the InterSystems IRIS decimal type, while the InterSystems IRIS decimal type supports more precision. The InterSystems IRIS decimal type has a precision of approximately 18.96 decimal digits (usually 19 digits but sometimes only 18 decimal digits of precision) while the $DOUBLE type usually has a precision around 15.95 decimal digits (or 53 binary digits). By default, InterSystems IRIS represents a numeric literal as a decimal floating-point number. However, if the numeric literal is larger than what can be represented in InterSystems IRIS decimal (larger than 9.223372036854775807E145) InterSystems IRIS automatically converts that numeric value to $DOUBLE representation.

A numeric value larger than 1.7976931348623157081E308 (308 or 309 digits) results in a <MAXNUMBER> error.

Because of the automatic conversion from decimal floating-point to binary floating-point, rounding behavior changes at 9.223372036854775807E145 (146 or 147 digits, depending on the integer). This is shown in the following examples:

ObjectScript

```
TRY {
  SET a=1
  FOR i=1:1:310 {SET a=a_1 WRITE i+1," digits = ",+a,! }
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SystemException") {
                WRITE "System exception",!
                WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
                WRITE "Location: ",exp.Location,!
                WRITE "Code: "
              }
            ELSE { WRITE "Some other type of exception",! RETURN }
            WRITE exp.Code,!
            WRITE "Data: ",exp.Data,!
            RETURN
}
```

ObjectScript

```
TRY {
  SET a=9
  FOR i=1:1:310 {SET a=a_9 WRITE i+1," digits = ",+a,! }
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SystemException") {
                WRITE "System exception",!
                WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
                WRITE "Location: ",exp.Location,!
                WRITE "Code: "
            }
            ELSE { WRITE "Some other type of exception",! RETURN }
            WRITE exp.Code,!
            WRITE "Data: ",exp.Data,!
            RETURN
}
```

You can represent a number longer than 309 digits as a numeric string. Because this value is stored as a string rather than a number, neither rounding nor the <MAXNUMBER> error apply:

ObjectScript

```
SET a="1"
FOR i=1:1:360 {SET a=a_"1" WRITE i+1," characters = ",a,! }
```

Exponents that would result in a number with more than the maximum permitted number of digits generate a <MAXNUMBER> error. The largest permitted exponent depends on the size of the number that is receiving the exponent. For a single-digit mantissa, the maximum exponent is 307 or 308.

For further details on large number considerations, see Numeric Computing in InterSystems Applications.

12.8 String-to-Number Conversion

Most of the operators introduced on this page interpret their operands as numeric values. This section describes how that interpretation is performed. First, here is the basic terminology:

- A *numeric string* is a string literal that consists entirely of numeric characters. For example, `"123"`, `"+123"`, `".123"`, `"++0007"`, `"-0"`.
- A *partially numeric string* is a string literal that begins with numeric symbols, followed by non-numeric characters. For example, `"3 blind mice"`, `"-12 degrees"`.
- A *non-numeric string* is a string literal that begins with a non-numeric character. For example, `" 123"`, `"the 3 blind mice"`, `"three blind mice"`.

12.8.1 Numeric Strings

When a numeric string or partially numeric string is used in an arithmetic expression, it is interpreted as a number. This numeric value is obtained by scanning the string from left to right to find the longest sequence of leading characters that can be interpreted as a numeric literal. The following characters are permitted:

- The digits 0 through 9.
- The PlusSign and MinusSign property values. By default these are the `+` and `-` characters, but are locale-dependent. Use the **%SYS.NLS.Format.GetFormatItem()** method to return the current settings.
- The DecimalSeparator property value. By default this is the `.` character, but is locale-dependent. Use the **%SYS.NLS.Format.GetFormatItem()** method to return the current setting.

- The letters `e`, and `E` may be included as part of a numeric string when in a sequence representing scientific notation, such as 4E3.

Note that the NumericGroupSeparator property value (the `,` character, by default) is *not* considered a numeric character. Therefore, the string `"123,456"` is a partially numeric string that resolves to the number `"123"`.

Numeric strings and partial numeric strings are converted to canonical form prior to arithmetic operations (such as addition and subtraction) and greater than/less than comparison operations (<, >, <=, >=). Numeric strings are *not* converted to canonical form prior to equality comparisons (=, '=), because these operators are also used for string comparisons.

The following example shows arithmetic comparisons of numeric strings:

Terminal

```
USER>WRITE "3" + 4
7
USER>WRITE "003.0" + 4
7
USER>WRITE "++--3" + 4
7
USER>WRITE "3 blind mice" + 4
7
```

The following example shows less than (<) comparisons of numeric strings:

Terminal

```
USER>WRITE "3" < 4
1
USER>WRITE "003.0" < 4
1
USER>WRITE "++--3" < 4
1
USER>WRITE "3 blind mice" < 4
1
```

The following example shows <= comparisons of numeric strings:

Terminal

```
USER>WRITE "4" <= 4
1
USER>WRITE "004.0" <= 4
1
USER>WRITE "++--4" <= 4
1
USER>WRITE "4 horsemen" <= 4
1
```

The following example shows equality comparisons of numeric strings. Non-canonical numeric strings are compared as character strings, not as numbers. Note that –0 is a non-canonical numeric string, and is therefore compared as a string, not a number:

Terminal

```
USER>WRITE "4" = 4.00
1
USER>WRITE "004.0" = 4
0
USER>WRITE "++--4" = 4
0
USER>WRITE "4 horsemen" = 4
0
USER>WRITE "-4" = -4
1
USER>WRITE "0" = 0
1
USER>WRITE "-0" = 0
0
USER>WRITE "-0" = -0
0
```

12.8.2 Non-Numeric Strings

If the leading characters of the string are not numeric characters, the string's numeric value is 0 for all arithmetic operations. For <, >, '>, <=, '<, and >= comparisons a non-numeric string is also treated as the number 0. Because the equal sign is used for both the numeric equality operator and the string comparison operator, string comparison takes precedence for = and '= operations. You can prepend the PlusSign property value (+ by default) to force numeric evaluation of a string; for example, `"+123"`. This results in the following logical values, when *x* and *y* are different non-numeric strings (for example x="Fred", y="Wilma").

x, y	x, x	+x, y	+x, +y	+x, +x
x=*y* is FALSE	*x*=*x* is TRUE	+*x*=*y* is FALSE	+*x*=+*y* is TRUE	+*x*=+*x* is TRUE
x'=*y* is TRUE	*x*'=*x* is FALSE	+*x*'=*y* is TRUE	+*x*'=+*y* is FALSE	+*x*'=+*x* is FALSE
x<*y* is FALSE	*x*<*x* is FALSE	+*x*<*y* is FALSE	+*x*<+*y* is FALSE	+*x*<+*x* is FALSE
x<=*y* is TRUE	*x*<=*x* is TRUE	+*x*<=*y* is TRUE	+*x*<=+*y* is TRUE	+*x*<=+*x* is TRUE

12.8.3 Extremely Large Numbers from Strings

Usually, a numeric string is converted to an ObjectScript Decimal value. However, with extremely large numbers (larger than 9223372036854775807E127) it is not always possible to convert a numeric string to a Decimal value. If converting a numeric string to its Decimal value would result in a <MAXNUMBER> error, InterSystems IRIS instead converts it to an IEEE Binary value. InterSystems IRIS performs the following operations in converting a numeric string to a number:

1. Convert numeric string to Decimal floating point number. If this would result in <MAXNUMBER> go to Step 2. Otherwise, return Decimal value as a canonical number.
2. Check the **$SYSTEM.Process.TruncateOverflow()** method boolean value. If 0 (the default) go to Step 3. Otherwise, return an overflow Decimal value (see method description).
3. Convert numeric string to IEEE Binary floating point number. If this would result in <MAXNUMBER> go to Step 4. Otherwise, return IEEE Binary value as a canonical number.
4. Check the **$SYSTEM.Process.IEEEError()** method boolean value. Depending on this value either return INF / -INF, or issue a <MAXNUMBER> error.

12.9 See Also

- ObjectScript Operators
- ObjectScript Functions
- Numeric Computing in InterSystems Applications

13 Strings in ObjectScript

This page provides an overview of strings in ObjectScript and ways of working with them.

Note: In InterSystems SQL, string literals are delimited with single quotation marks, for example, `'sample literal string'` instead of double quotation marks as used in ObjectScript. This is important to remember when you write a mix of ObjectScript and SQL code.

13.1 String Literals

In ObjectScript, a string literal is a set of zero or more characters delimited by double quotation marks, for example:

```
"sample literal string"
```

The value can contain any characters, including whitespace characters, control characters, and Unicode characters that cannot be typed.

To include a double quotation mark within a string, use two double quotation marks with no space between them:

```
"sample literal string with ""quoted"" text"
```

There is a maximum permitted length (see String Length Limit).

The following example shows a string of 8-bit characters, a string of 16-bit Unicode characters (Greek letters), and a combined string:

ObjectScript

```
  DO AsciiLetters
  DO GreekUnicodeLetters
  DO CombinedAsciiUnicode
  RETURN
AsciiLetters()
  SET a="abc"
  WRITE a
  WRITE !,"the length of string a is ",$LENGTH(a)
  ZZDUMP a
  QUIT
GreekUnicodeLetters()
  SET b=$CHAR(945)_$CHAR(946)_$CHAR(947)
  WRITE !!,b
  WRITE !,"the length of string b is ",$LENGTH(b)
  ZZDUMP b
  QUIT
CombinedAsciiUnicode()
  SET c=a_b
  WRITE !!,c
  WRITE !,"the length of string c is ",$LENGTH(c)
  ZZDUMP c
  QUIT
```

13.2 Non-Printing Characters and Unicode Characters

When you create a string, sometimes you need to include characters that cannot be typed. For these, you use the **$CHAR** function. Given an integer, **$CHAR** returns the corresponding ASCII or Unicode character. Common uses:

- **$CHAR(9)** is a tab.
- **$CHAR(10)** is a line feed.
- **$CHAR(13)** is a carriage return.
- **$CHAR(13,10)** is a carriage return and line feed pair.

The function **$ASCII** returns the ASCII value of the given character.

Similarly, you can use the **$CHAR** function to specify Unicode characters that cannot be typed (depending on your keyboard), as shown in the following example:

ObjectScript

```
SET greekstr=$CHAR(952,945,955,945,963,963,945)
WRITE greekstr
```

Note: How non-printing characters display is determined by the display device. For example, the Terminal differs from browser display of the linefeed character, and other positioning characters. In addition, different browsers display the positioning characters $CHAR(11) and $CHAR(12) differently.

13.3 Null Strings and the Null Character

An empty string, represented by two quotation mark characters (`""`), is known as a null string. A null string is considered to be a defined value; that is, if a variable is set equal to the empty string, the variable is considered defined. An empty string has a length of 0.

The null string is *not* the same as a string consisting of the ASCII null character ($CHAR(0)), as shown in the following example:

ObjectScript

```
SET x=""
WRITE "string=",x," length=",$LENGTH(x)," defined=",$DATA(x)
ZZDUMP x
SET y=$CHAR(0)
WRITE !!,"string=",y," length=",$LENGTH(y)," defined=",$DATA(y)
ZZDUMP y
```

13.4 String Concatenation

You can concatenate two strings into a single string using the concatenate operator:

ObjectScript

```
SET a = "Inter"
SET b = "Systems"
SET string = a_b
WRITE string
```

For InterSystems IRIS encoded strings — bit strings, list structure strings, and JSON strings— there are limitations on the concatenate operator. For further details, see Concatenate Encoded Strings.

Some additional considerations apply when concatenating numbers; see Concatenating Numbers.

13.5 String Equality

You can use the equals (=) and does not equal ('=) operators to compare two strings. String equality comparisons are case-sensitive. Exercise caution when using these operators to compare a string to a number, because this comparison is a string comparison, not a numeric comparison. Therefore only a string containing a number in canonical form is equal to its corresponding number. ("-0" is not a canonical number.) This is shown in the following example:

ObjectScript

```
 WRITE "Fred" = "Fred",!  // TRUE
 WRITE "Fred" = "FRED",!  // FALSE
 WRITE "-7" = -007.0,!    // TRUE
 WRITE "-007.0" = -7,!    // FALSE
 WRITE "0" = -0,!         // TRUE
 WRITE "-0" = 0,!         // FALSE
 WRITE "-0" = -0,!        // FALSE
```

Because string equality comparisons are case-sensitive, depending on the use case, it may be appropriate to first use the **$ZCONVERT** function to convert the strings to all uppercase letters or all lowercase letters. This functionj does not affect non-letter characters. A few letters only have a lowercase letter form. For example, the German eszett ($CHAR(223)) is only defined as a lowercase letter. Converting it to an uppercase letter results in the same lowercase letter. For this reason, when converting alphanumeric strings to a single letter case it is always preferable to convert to lowercase.

The <, >, <=, or >= operators cannot be used to perform a string comparison. These operators treat strings as numbers and always perform a numeric comparison.

13.6 Other String Relational Operators

In addition to the equals and does not equal operators, ObjectScript provides additional operators that interpret their operands as strings. You can precede any of them with the NOT logical operator (') to obtain the negation of the logical result. ObjectScript provides the following string relational operators:

Contains ([)

Tests whether the sequence of characters in the right operand is a substring of the left operand. If the left operand contains the character string represented by the right operand, the result is TRUE (1). If the left operand does not contain the character string represented by the right operand, the result is FALSE (0). If the right operand is the null string, the result is always TRUE.

For example:

Terminal

```
USER>SET L="Steam Locomotive"

USER>SET S="Steam"

USER>WRITE L[S
1
```

See the Contains ([) and Does Not Contain ('[) reference pages.

Follows (])

Tests whether the characters in the left operand come after the characters in the right operand *in ASCII collating sequence*. Follows tests both strings starting with the left most character in each.

For example:

Terminal

```
USER>WRITE "LAMPOON"]"LAMP"
1
```

See the Follows (]) and Not Follows (']) reference pages.

Sorts After (]])

Tests whether the left operand sorts after the right operand *in numeric subscript collation sequence*. In numeric collation sequence, the null string collates first, followed by canonical numbers in numeric order with negative numbers first, zero next, and positive numbers, followed lastly by nonnumeric values.

For example:

Terminal

```
USER>WRITE 122]]2
1
```

See the Sorts After (]]) and Not Sorts After (']]) reference pages.

13.7 Pattern Match Operator (?)

The ObjectScript pattern match operator tests whether the characters in its left operand are correctly specified by the pattern in its right operand.

For example, the following tests a couple of strings to see if they are valid U.S. Social Security Numbers:

Terminal

```
USER>SET test1="123-45-6789"

USER>SET test2="123-XX-6789"

USER>WRITE test1 ? 3N1"-"2N1"-"4N
1
USER>WRITE test2 ? 3N1"-"2N1"-"4N
0
```

See the Pattern Match (?) reference page.

Note: ObjectScript also supports regular expressions, a pattern match syntax supported (with variants) by many software vendors. Regular expressions can be used with the **$LOCATE** and **$MATCH** functions, and with methods of the %Regex.Matcher class. For details, see the Regular Expressions reference page.

These pattern match systems are wholly separate and use different syntaxes with different patterns and flags.

13.8 Commonly Used String Functions

The most commonly used ObjectScript functions for operating on strings include:

- The **$LENGTH** function returns the number of characters in a string: For example, the code:

  ObjectScript

  ```
   WRITE $LENGTH("How long is this?")
  ```

  returns 17, the length of a string.

- **$JUSTIFY** returns a right-justified string, padded on the left with spaces (and can also perform operations on numeric values). For example, the code:

  ObjectScript

  ```
   WRITE "one",!,$JUSTIFY("two",8),!,"three"
  ```

  justifies string `two` within eight characters and returns:

  ```
  one
       two
  three
  ```

- **$ZCONVERT** converts a string from one form to another. It supports both case translations (to uppercase, to lowercase, or to title case) and encoding translation (between various character encoding styles). For example, the code:

  ObjectScript

  ```
   WRITE $ZCONVERT("cRAZy cAPs","t")
  ```

  returns:

  ```
  CRAZY CAPS
  ```

- The **$FIND** function searches for a substring of a string, and returns the position of the character *following* the substring. For example, the code:

  ObjectScript

  ```
   WRITE $FIND("Once upon a time...", "upon")
  ```

  returns 10 character position immediately following “upon.”

- The **$TRANSLATE** function performs a character-by-character replacement within a string. For example, the code:

  ObjectScript

  ```
   SET text = "11/04/2008"
   WRITE $TRANSLATE(text,"/","-")
  ```

replaces the date's slashes with hyphens.

- The **$REPLACE** function performs string-by-string replacement within a string; the function does not change the value of the string on which it operates. For example, the following code performs two distinct operations:

  ObjectScript

  ```
  SET text = "green leaves, brown leaves"
  WRITE text,!
  WRITE $REPLACE(text,"leaves","eyes"),!
  WRITE $REPLACE(text,"leaves","hair",15),!
  WRITE text,!
  ```

  In the first call, **$REPLACE** replaces the string `leaves` with the string `eyes`. In the second call, **$REPLACE** discards all the characters prior to the fifteenth character (specified by the fourth argument) and replaces the string `leaves` with the string `hair`. The value of the *text* string is not changed by either **$REPLACE** call.

- The **$EXTRACT** function, which returns a substring from a specified position in a string. For example, the code:

  ObjectScript

  ```
  WRITE $EXTRACT("Nevermore"),$EXTRACT("prediction",5),$EXTRACT("xon/xoff",1,3)
  ```

  returns three strings. The one-argument form returns the first character of the string; the two-argument form returns the specified character from the string; and the three-argument form returns the substring beginning and ending with specified characters, inclusive. In the example above, there are no line breaks, so the return value is:

  ```
  Nixon
  ```

13.9 See Also

- ObjectScript Operators
- ObjectScript Functions

14 Boolean Values in ObjectScript

This page describes Boolean values in ObjectScript, as well as operators for working with them.

14.1 Introduction

Unlike some other languages, ObjectScript does not provide specialized representations of Boolean literal values. Instead:

- The value 1 and any expression that be interpreted as a nonzero numeric value is considered true. See String-to-Number Conversions for information on how strings are interpreted.
- The value 0 and any expression that can be interpreted as 0 or that has no numeric interpretation is considered false.

Also, operators that *interpret* operands as Boolean values return 1 for true or 0 for false.

When you need a simple representation of Boolean values, for simplicity, use 1 for true, and 0 for false.

14.2 Uses

Boolean values are most commonly used with:

- The IF command
- The $SELECT function
- Postconditional Expressions

14.3 Logical Operators

The following operators always treat their operands as Boolean values.

Not (')

Not inverts the truth value of the Boolean operand. If the operand is TRUE (1), Not gives it a value of FALSE (0). If the operand is FALSE (0), Not gives it a value of TRUE (1).

For example:

Terminal

```
USER>SET x=0

USER>WRITE 'x
1
```

See the Not (') reference page.

And (& or &&)

And tests whether both its operands have a truth value of TRUE (1). If both operands are TRUE (that is, have nonzero values when evaluated numerically), ObjectScript produces a value of TRUE (1). Otherwise, ObjectScript produces a value of FALSE (0).

There are two forms to And:

- The & operator evaluates both operands and returns a value of FALSE (0) if either operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).
- The && operator evaluates the left operand and returns a value of FALSE (0) if it evaluates to a value of zero. Only if the left operand is nonzero does the && operator then evaluate the right operand. It returns a value of FALSE (0) if the right operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).

For example.

Terminal

```
USER>SET A=-4,B=1

USER>WRITE A&&B
1
```

See the And (& or &&) and Not And (NAND) ('&) reference pages.

Or (! or ||)

Or produces a result of TRUE (1) if either operand has a value of TRUE or if both operands have a value of TRUE (1). Or produces a result of FALSE (0) only if both operands are FALSE (0).

There are two forms to Or:

- The ! operator evaluates both operands and returns a value of FALSE (0) if both operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).
- The || operator evaluates the left operand. If the left operand evaluates to a nonzero value, the || operator returns a value of TRUE (1) without evaluating the right operand. Only if the left operand evaluates to zero does the || operator then evaluate the right operand. It returns a value of FALSE (0) if the right operand also evaluates to a value of zero. Otherwise it returns a value of TRUE (1).

For example:

Terminal

```
USER>SET A=5,B=7

USER>WRITE A!B
1
USER>WRITE A||B
1
```

See the Or (! or ||) and Not Or (NOR) ('!) reference pages.

14.4 Precedence and Logical Operators

Because ObjectScript performs a strict left-to-right evaluation of operators, logical comparisons involving other operators must use parentheses to group operations to achieve the desired precedence. For example, you would expect the logical Or (!) test in the following program to return TRUE (1):

ObjectScript

```
  SET x=1,y=0
  IF x=1 ! y=0 {WRITE "TRUE"}
  ELSE {WRITE "FALSE" }
  // Returns 0 (FALSE), due to evaluation order
```

However, to properly perform this logical comparison, you must use parentheses to nest the other operations. The following example gives the expected results:

ObjectScript

```
  SET x=1,y=0
  IF (x=1) ! (y=0) {WRITE "TRUE"}
  ELSE {WRITE "FALSE" }
  // Returns 1 (TRUE)
```

14.5 Combining Boolean Values

You can combine multiple Boolean logical expressions by using logical operators. Like all InterSystems IRIS expressions, they are evaluated in strict left-to-right order. There are two types of logical operators: regular logical operators (& and !) and short-circuit logical operators (&& and ||).

When regular logical operators are used to combine logical expressions, InterSystems IRIS evaluates all of the specified expressions, even when the Boolean result is known before all of the expressions have been evaluated. This assures that all expressions are valid.

When short-circuit logical operators are used to combine logical expressions, InterSystems IRIS evaluates only as many expressions as are needed to determine the Boolean result. For example, if there are multiple AND tests, the first expression that returns 0 determines the overall Boolean result. Any logical expressions to the right of this expression are not evaluated. This allows you to avoid unnecessary time-consuming expression evaluations.

Some commands allow you to specify a comma-separated list as an argument value. In this case, InterSystems IRIS handles each listed argument like an independent command statement. Therefore, `IF x=7,y=4,z=2` is parsed as `IF x=7 THEN IF y=4 THEN IF z=2`, which is functionally identical to the short-circuit logical operators statement `IF (x=7)&&(y=4)&&(z=2)`.

In the following example, the IF test uses a regular logical operator (&). Therefore, all functions are executed even though the first function returns 0 (FALSE) which automatically makes the result of the entire expression FALSE:

ObjectScript

```
LogExp
 IF $$One() & $$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

In the following example, the IF test uses a short-circuit logical operator (&&). Therefore, the first function is executed and returns 0 (FALSE) which automatically makes the result of the entire expression FALSE. The second function is not executed:

ObjectScript

```
LogExp
 IF $$One() && $$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

In the following example, the IF test specifies comma-separated arguments. The comma is not a logical operator, but has the same effect as specifying the short-circuit && logical operator. The first function is executed and returns 0 (FALSE) which automatically makes the result of the entire expression FALSE. The second function is not executed:

ObjectScript

```
LogExp
 IF $$One(),$$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

14.6 See Also

- ObjectScript Operators

15 Lists in ObjectScript

This topic provides an overview of the ObjectScript native list format and alternatives. One simple alternative is the delimited string. There are class-based alternatives as well.

15.1 Native List Format

ObjectScript provides a *native list format*. This format is sometimes called *$LIST format*, because the $LIST function is used to work with these lists.

The only supported way to work with the native list format is to use the ObjectScript list functions. The internal structure of this kind of list is not documented and is subject to change without notice.

In class definitions, if you want a property to use the native list format, declare the property type as %Library.List or the short name %List.

15.2 List Functions

The ObjectScript native list format consists of an encoded list of substrings, known as elements. These lists can only be handled using the following list functions:

- List creation:
  - $LISTBUILD creates a list by specifying each element as a parameter value.
  - $LISTFROMSTRING creates a list by specifying a string that contains delimiters. The function uses the delimiter to divide the string into elements.
  - $LIST creates a list by extracting it as a sublist from an existing list.
- List data retrieval:
  - $LIST returns a list element value by position. It can count positions from the beginning or the end of the list.
  - $LISTNEXT returns list element values sequentially from the beginning of the list. While both **$LIST** and **$LISTNEXT** can be used to sequentially return elements from a list, **$LISTNEXT** is significantly faster when returning a large number of list elements.
  - $LISTGET returns a list element value by position, or returns a default value.

  - $LISTTOSTRING returns all of the element values in a list as a delimited string.

- List manipulation:
  - SET $LIST inserts, updates, or deletes elements in a list. **SET $LIST** replaces a list element or a range of list elements with one or more values. Because **SET $LIST** can replace a list element with more than one element, you can use it to insert elements into a list. Because **SET $LIST** can replace a list element with a null string, you can use it to delete a list element or a range of list elements.

- List evaluation:
  - $LISTVALID determines if a string is a valid list.
  - $LISTLENGTH determines the number of elements in a list.
  - $LISTDATA determines if a specified list element contains data.
  - $LISTFIND determines if a specified value is found in a list, returning the list position.
  - $LISTSAME determines if two lists are identical.

Because a list is an encoded string, InterSystems IRIS treats lists slightly differently than standard strings. Therefore, you should not use standard string functions on lists. Further, using most list functions on a standard string generates a <LIST> error.

The following procedure demonstrates the use of the various list functions:

ObjectScript

```
ListTest() PUBLIC {
    // set values for list elements
    SET Addr="One Memorial Drive"
    SET City="Cambridge"
    SET State="MA"
    SET Zip="02142"

    // create list
    SET Mail = $LISTBUILD(Addr,City,State,Zip)

    // get user input
    READ "Enter a string: ",input,!,!

    // if user input is part of the list, print the list's content
    IF $LISTFIND(Mail,input) {
        FOR i=1:1:$LISTLENGTH(Mail) {
            WRITE $LIST(Mail,i),!
        }
     }
}
```

This procedure demonstrates several notable aspects of lists:

- **$LISTFIND** only returns 1 (True) if the value being tested matches the list item exactly.
- **$LISTFIND** and **$LISTLENGTH** are used in expressions.

For more detailed information on list functions see the corresponding reference pages in the *ObjectScript Reference*.

15.3 Sparse Lists and Sublists

A function that adds an element value to a list by position will add enough list elements to place the value in the proper position. For example:

ObjectScript

```
  SET $LIST(Alphalist,1)="a"
  SET $LIST(Alphalist,20)="t"
  WRITE $LISTLENGTH(Alphalist)
```

Because the second **$LIST** in this example creates list element 20, **$LISTLENGTH** returns a value of 20. However, elements 2 through 19 do not have values set. Hence, if you attempt to display any of their values, you will receive a <NULL VALUE> error. You can use **$LISTGET** to avoid this error.

An element in a list can itself be a list. To retrieve a value from a sublist such as this, nest **$LIST** function calls, as in the following code:

ObjectScript

```
  SET $LIST(Powers,2)=$LISTBUILD(2,4,8,16,32)
  WRITE $LIST($LIST(Powers,2),5)
```

This code returns 32, which is the value of the fifth element in the sublist contained by the second element in the *Powers* list. (In the *Powers* list, the second item is a sublist of two raised to the first through fifth powers, so that the first item in the sublist is two to the first power, and so on.)

15.4 List Compression

The ListFormat setting controls whether Unicode strings should be compressed when stored in a $LIST encoded string. The default is to not compress. Compressed format is automatically handled by InterSystems IRIS. Do not pass compressed lists to external clients, such as Java or C#, without verifying that they support the compressed format.

The per-process behavior can be controlled using the **ListFormat()** method of the %SYSTEM.Process class.

The system-wide default behavior can be established by setting the *ListFormat* property of the Config.Miscellaneous class or the InterSystems IRIS Management Portal, as follows: from **System Administration**, select **Configuration**, **Additional Settings**, **Compatibility**.

15.5 Delimited Strings as an Alternative

As a simple alternative to the native list format, you can use a delimiter-separated string as a list. In this case, you generally use the following functions:

- **$PIECE** — Returns a specific piece of a string based on a specified delimiter. It can also return a range of pieces, as well as multiple pieces from a single string, based on multiple delimiters.
- **$LENGTH** — Returns the number of pieces in a string based on a specified delimiter.

The **$PIECE** function provides uniquely important functionality because it allows you to use a single string that contains multiple substrings, with a special delimiter character (such as ^) to separate them. The large string acts as a record, and the substrings are its fields.

The syntax for **$PIECE** is:

ObjectScript

```
 WRITE $PIECE("ListString","QuotedDelimiter",ItemNumber)
```

where *ListString* is a quoted string that contains the full record being used; *QuotedDelimiter* is the specified delimited, which must appear in quotes; and *ItemNumber* is the specified substring to be returned. For example, to display the second item in the following space-delimited list, the syntax is:

ObjectScript

```
 WRITE $PIECE("Kennedy Johnson Nixon"," ",2)
```

which returns `Johnson`.

You can also return multiple members of the list, so that the following:

ObjectScript

```
 WRITE $PIECE("Nixon***Ford***Carter***Reagan","***",1,3)
```

returns `Nixon***Ford***Carter`. Note that both values must refer to actual substrings and the third argument (here 1) must be a smaller value than that of the fourth argument (here 3).

The delimiter can be anything you choose, such as with the following list:

ObjectScript

```
 SET x = $PIECE("Reagan,Bush,Clinton,Bush,Obama",",",3)
 SET y = $PIECE("Reagan,Bush,Clinton,Bush,Obama","Bush",2)
 WRITE x,!,y
```

which returns

```
Clinton
,Clinton,
```

In the first case, the delimiter is the comma; in the second, it is the string `Bush`, which is why the returned string includes the commas. To avoid any possible ambiguities related to delimiters, use the list-related functions, described in the next section.

15.5.1 Advanced $PIECE Features

A call to **$PIECE** that sets the value of a delimited element in a list will add enough list items so that it can place the substring as the proper item in an otherwise empty list. For instance, suppose some code sets the first, then the fourth, then the twentieth item in a list,

ObjectScript

```
 SET $PIECE(Alphalist, "^", 1) = "a"
 WRITE "First, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
 SET $PIECE(Alphalist, "^", 4) = "d"
 WRITE "Then, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
 SET $PIECE(Alphalist, "^", 20) = "t"
 WRITE "Finally, the length of the list is ",$LENGTH(Alphalist,"^"),".",!
```

The **$LENGTH** function returns a value of 1, then 4, then 20, since it creates the necessary number of delimited items. However, items 2, 3, and 5 through 19 do not have values set. Hence, if you attempt to display any of their values, nothing appears.

A delimited string item can also contain a delimited string. To retrieve a value from a sublist such as this, nest **$PIECE** function calls, as in the following code:

ObjectScript

```
SET $PIECE(Powers, "^", 1) = "1::1::1::1::1"
SET $PIECE(Powers, "^", 2) = "2::4::8::16::32"
SET $PIECE(Powers, "^", 3) = "3::9::27::81::243"
WRITE Powers,!
WRITE $PIECE( $PIECE(Powers, "^", 2), "::", 3)
```

This code returns two lines of output: the first is the string *Powers*, including all its delimiters; the second is 8, which is the value of the third element in the sublist contained by the second element in *Powers*. (In the *Powers* list, the *n*th item is a sublist of two raised to the first through fifth powers, so that the first item in the sublist is *n* to the first power, and so on.)

For more details, see $PIECE.

15.6 Lists and Delimited Strings Compared

The native list format provides the following advantages, when compared to delimited strings:

- The native list format does not require a designated delimiter. Though the **$PIECE** function allows you to manage a string containing multiple data items, it depends on setting aside a character (or character string) as a dedicated delimiter. When using delimiters, there is always the chance that one of the data items will contain the delimiter character(s) as data, which will throw off the positions of the pieces in the delimited string. A list is useful for avoiding delimiters altogether, and thus allowing any character or combination of characters to be entered as data.
- Data elements can be retrieved faster from a list (using **$LIST** or **$LISTNEXT**) than from a delimited string (using **$PIECE**). For sequential data retrieval, **$LISTNEXT** is significantly faster than **$LIST**, and both are significantly faster than **$PIECE**.

A delimited string provides different advantages, when compared to the native list format:

- A delimited string allows you to more flexibly search the contents of data, using the **$FIND** function. Because **$LISTFIND** requires an exact match, you cannot search for partial substrings in lists. Hence, in the example above, using **$LISTFIND** to search for the string `One` in the Mail list return 0 (indicating failure), even though the address `One Memorial Drive` begins with the characters `One`.
- Because a delimited string is a standard string, you can use all of the standard string functions on it. Because a native list is an encoded string, you can only use $List functions on it.

15.7 Class-Based Lists

Rather than the native list format or delimited strings, you can use list classes that use different structure and that provide an object-based API. InterSystems IRIS provides a set of classes you can use as list-form class properties, and another set of classes you can use for standalone lists.

15.8 See Also

- Working with Collection Classes

16 Native Bit Strings in ObjectScript

ObjectScript provides a native bit string format and functions to work with this format.

A bit string represents a logical set of numbered bits with boolean values. Bits in a string are numbered starting with bit number 1. Any numbered bit that has not been explicitly set to boolean value 1 evaluates as 0. Therefore, referencing any numbered bit beyond those explicitly set returns a bit value of 0.

- Bit values can only be set using the bit string functions $BIT and $BITLOGIC.
- Bit values can only be accessed using the bit string functions $BIT, $BITLOGIC, and $BITCOUNT.

A bit string has a logical length, which is the highest bit position explicitly set to either 0 or 1. This logical length is only accessible using the $BITCOUNT function, and usually should not be used in application logic. To the bit string functions, an undefined global or local variable is equivalent to a bitstring with any specified numbered bit returning a bit value 0, and a **$BITCOUNT** value of 0.

A bit string is stored as a normal ObjectScript string with an internal format. This internal string representation is not accessible with the bit string functions. Because of this internal format, the string length of a bit string is not meaningful in determining anything about the number of bits in the string.

Because of the bit string internal format, you cannot use the concatenate operator with bit strings. Attempting to do so results in an <INVALID BIT STRING> error.

Two bit strings in the same state (with the same boolean values) may have different internal string representations, and therefore string representations should not be inspected or compared in application logic.

To the bit string functions, a bitstring specified as an undefined variable is equivalent to a bitstring with all bits 0, and a length of 0.

Unlike an ordinary string, a bit string treats the empty string and the character $CHAR(0) to be equivalent to each other and to represent a 0 bit. This is because **$BIT** treats any non-numeric string as 0. Therefore:

ObjectScript

```
  SET $BIT(bstr1,1)=""
  SET $BIT(bstr2,1)=$CHAR(0)
  SET $BIT(bstr3,1)=0
  IF $BIT(bstr1,1)=$BIT(bstr2,1) {WRITE "bitstrings are the same"} ELSE {WRITE "bitstrings different"}

  WRITE $BITCOUNT(bstr1),$BITCOUNT(bstr2),$BITCOUNT(bstr3)
```

A bit set in a global variable during a transaction will be reverted to its previous value following transaction rollback. However, rollback does not return the global variable bit string to its previous string length or previous internal string representation. Local variables are not reverted by a rollback operation.

A logical bitmap structure can be represented by an array of bit strings, where each element of the array represents a "chunk" with a fixed number of bits. Since undefined is equivalent to a chunk with all 0 bits, the array can be sparse, where array elements representing a chunk of all 0 bits need not exist at all. For this reason, and due to the rollback behavior above, application logic should avoid depending on the length of a bit string or the count of 0-valued bits accessible using **$BITCOUNT(str)** or **$BITCOUNT(str,0)**.

17 Date and Time Values

This page provides an overview of date and time values in ObjectScript.

17.1 Introduction

ObjectScript has no built-in date type; instead it includes a number of functions for operating on and formatting date values represented as strings. These date formats include:

Table 17–1: Date Formats

Format	Description
$HOROLOG	This is the format returned by the $HOROLOG ($H) special variable. It is a string containing two comma-separated integers: the first is the number of days since December 31, 1840; the second is the number of seconds since midnight of the current day. $HOROLOG does not support fractional seconds. The $NOW function provides $HOROLOG-format dates with fractional seconds. InterSystems IRIS provides a number of functions for formatting and validating dates in $HOROLOG format.
ODBC Date	This is the format used by ODBC and many other external representations. It is a string of the form: "YYYY-MM-DD HH:MM:SS". ODBC date values will collate; that is, if you sort data by ODBC date format, it will automatically be sorted in chronological order.
Locale Date	This is the format used by the current locale. Locales differ in how they format dates as follows: "American" dates are formatted mm/dd/yyyy (dateformat 1). "European" dates are formatted dd/mm/yyyy (dateformat 4). All locales use dateformat 1 except the following — csyw, deuw, engw, espw, eurw, fraw, itaw, mitw, ptbw, rusw, skyw, svnw, turw, ukrw — which use dateformat 4. American dates use a period (.) as a decimalseparator for fractional seconds. European dates use a comma (,) as a decimalseparator for fractional seconds, except the following — engw, eurw, skyw — which use a period. All locales use a slash (/) as the dateseparator character, except the following, which use a period (.) as the dateseparator character — Czech (csyw), Russian (rusw), Slovak (skyw), Slovenian (svnw), and Ukrainian (ukrw).
System Time	This is the format returned by the $ZHOROLOG ($ZH) special variable. It is a floating point number containing the number of seconds (and parts thereof) that the system has been running. Stopping and restarting InterSystems IRIS resets this number. Typically this format is used for timing and testing operations.

The following example shows how you can use the different date formats:

ObjectScript

```
 SET now = $HOROLOG
 WRITE "Current time and date ($H): ",now,!

 SET odbc = $ZDATETIME(now,3)
 WRITE "Current time and date (ODBC): ",odbc,!

 SET ldate = $ZDATETIME(now,-1)
 WRITE "Current time and date in current locale format: ",ldate,!

 SET time = $ZHOROLOG
 WRITE "Current system time ($ZH): ",time,!
```

17.2 Date and Time Conversions

ObjectScript includes functions for converting date and time values.

- Given a date in $H format, the function **$ZDATE** returns a string that represents the date in your specified format.

For example:

```
TESTNAMESPACE>WRITE $ZDATE($HOROLOG,3)
2010-12-03
```

- Given a date and time in $H format, the function **$ZDATETIME** returns a string that represents the date and time in your specified format.

  For example:

  ```
  TESTNAMESPACE>WRITE $ZDATETIME($HOROLOG,3)
  2010-12-03 14:55:48
  ```

- Given string dates and times in other formats, the functions **$ZDATEH** and **$ZDATETIMEH** convert those to $H format.
- The functions **$ZTIME** and **$ZTIMEH** convert times from and to $H format.

17.3 Details of the $H Format

The $H format is a pair of numbers separated by a comma. For example: `54321,12345`

- The first number is the number of days since December 31st, 1840. That is, day number 1 is January 1st, 1841. This number is always an integer.
- The second number is the number of seconds since midnight on the given day.

  Some functions, such as **$NOW()**, provide a fractional part.

For additional details, including an explanation of the starting date, see $HOROLOG.

18 Macros and Include Files

This page describes how to define and use macros and include files (which contain macros). InterSystems IRIS® data platform provides system macros that you can use as well.

Important: The phrase *include file* is used for historical reasons but unfortunately also creates some confusion. In InterSystems IRIS, an include file is not actually a separate standalone file in the operating system. As with classes and routines, an include file is a unit of code stored within an InterSystems IRIS database.

Your IDE provides an option for creating an include file, and will store the code correctly in the database — the same as with any other code element. Similarly, if the IDE is connected to a source control system, each code element is projected to an external file that is managed via source control.

18.1 Macro Basics

A *macro* is a convenient substitution that you can define and use as follows:

1. You define the macro via special syntax, typically the #define directive. For example:

   ObjectScript

   ```
   #define StringMacro "Hello, World!"
   ```

   This syntax defines a macro called `StringMacro`. Note that macro names are case-sensitive.

2. Later, you invoke the macro with the syntax $$$*macroname*, for example:

   ObjectScript

   ```
     write $$$StringMacro
   ```

   The previous is equivalent to the following:

   ObjectScript

   ```
     write "Hello, World!"
   ```

The substitution occurs when the code (a class or routine) is compiled. Specifically, the class or routine itself is unchanged, but the generated .INT code shows the substitutions. (For a fuller picture of how code is compiled, see How These Code Elements Work Together.)

Remember that macros are text substitutions. After the substitution is performed, the resulting statement must be syntactically correct. Therefore, the macro defining an expression should be invoked in a context requiring an expression; the macro for a command and its argument can stand as an independent line of ObjectScript; and so on.

18.2 Include File Basics

Typically, you define macros within an *include file*, which you then include within other code, which enables that code to refer to the macros. This works as follows:

1. An *include file* is a specific kind of unit of code stored in the database. The following shows a partial example:

   ObjectScript

   ```
   #; Optional comment lines
   #define RELEASEID $GET(^MyGlobal("ReleaseID"),"")
   #define RELEASENUMBER $GET(^MyGlobal("ReleaseNumber"),"")
   #define PRODUCT $GET(^MyGlobal("Product"),"")
   #define LOCALE $GET(^MyGlobal("Locale"),"en-us")
   ```

   Notice that each line is either a comment line or starts with a #define directive. Blank lines are also permitted. There are alternatives to #define that enable you to define more complex macros; these are discussed elsewhere in more detail.

   In the typical scenario, you create an include file in your IDE and save it with a specific name, such as `MyMacros`.

2. Within a class or routine that needs to use the macros, include the include file. For example:

   Class Definition

   ```
   include MyMacros

   Class MyPackage.MyClass {

   //
   }
   ```

   In this example, the name of the include file is `MyMacros`.

   This step makes macros of `MyMacros` available for use within the class or routine.

   For all the syntax variations, which are different for routines, see Including Include Files.

3. Within that same class or routine, use the syntax $$$*macroname* to refer to the macro. For example:

   ObjectScript

   ```
    set title=$$$PRODUCT_" "_$$$RELEASENUMBER
   ```

Note: In running text, it is common to append `.inc` to the include file name; for example, a set of useful system macros are defined in the %occStatus.inc and %occMessages.inc include files.

18.3 Defining Macros

In their most basic form, macros are created with a #define directive as shown in Macro Basics.

There are additional directives that enable you to define macros that accept arguments and that support more complex scenarios. Also you can use ##continue to continue a #define directive to the next line. See Preprocessor Directives Reference for more.

This section provides information on where you can define macros, what macro definitions can contain, the rules that macro names must follow, use of whitespace in macros, and macro comments.

18.3.1 Where to Define Macros

You can define macros in the following locations, each of which affects the availability of the macros:

- You can define macros in an include file. In this case, the macros are available within any code that includes the necessary include file.

  Note that when a class includes an include file, any subclass of that class automatically includes the same include file.
- You can define macros within a method. In this case, the macros are available within that method.
- You can define macros within a routine. In this case, the macros are available within that routine.

18.3.2 Allowed Macro Definitions

Supported functionality includes:

- String substitutions, as demonstrated above.
- Numeric substitutions:

  ObjectScript

  ```
  #define NumberMacro 22
  ```

  ObjectScript

  ```
  #define 25M ##expression(25*1000*1000)
  ```

  As is typical in ObjectScript, the definition of the numeric macro does not require quoting the number, while the string must be quoted in the string macro’s definition.
- Variable substitutions:

  ObjectScript

  ```
  #define VariableMacro Variable
  ```

  Here, the macro name substitutes for the name of a variable that is already defined. If the variable is not defined, there is an <UNDEFINED> error.
- Command and argument invocations:

  ObjectScript

  ```
  #define CommandArgumentMacro(%Arg) WRITE %Arg,!
  ```

  Macro argument names must start with the % character, such as the *%Arg* argument above. Here, the macro invokes the **WRITE** command, which uses the *%Arg* argument.
- Use of functions, expressions, and operators:

ObjectScript

```
#define FunctionExpressionOperatorMacro ($ZDate(+$Horolog))
```

Here, the macro as a whole is an expression whose value is the return value of the **$ZDate** function. **$ZDate** operates on the expression that results from the operation of the `+` operator on the system time, which the system variable **$Horolog** holds. As shown above, it is a good idea to enclose expressions in parentheses so that they minimize their interactions with the statements in which they are used.

- References to other macros:

  ObjectScript

  ```
  #define ReferenceOtherMacroMacro WRITE $$$ReferencedMacro
  ```

  Here, the macro uses the expression value of another macro as an argument to the **WRITE** command.

  Note: If one macro refers to another, the referenced macro must appear on a line of code that is compiled before the referencing macro.

18.3.3 Macro Naming Conventions

- The first character must be an alphanumeric character or the percent character (%).
- The second and subsequent characters must be alphanumeric characters. A macro name may not include spaces, underscores, hyphens, or other symbol characters.
- Macro names are case-sensitive.
- Macro names can be up to 500 characters in length.
- Macro names can contain Japanese ZENKAKU characters and Japanese HANKAKU Kana characters. For further details, refer to the “Pattern Codes” table in Pattern Match Operator.
- Macro names should not begin with `ISC`, because `ISC`*name*.inc files are reserved for system use.

18.3.4 Macro Whitespace Conventions

- By convention, a macro directive is not indented and appears in column 1. However, a macro directive may be indented.
- One or more spaces may follow a macro directive. Within a macro, any number of spaces may appear between macro directive, macro name, and macro value.
- A macro directive is a single-line statement. The directive, macro name, and macro value must all appear on the same line. You can use ##continue to continue a macro directive to the next line.
- #if and #elseIf directives take a test expression. This test expression may not contain any spaces.
- An #if expression, an #elseIf expression, the #else directive, and the #endif directive all appear on their own line. Anything following one of these directives on the same line is considered a comment and is not parsed.

18.3.5 Macro Comments and Typeahead Assistance

Macros can include comments, which are passed through as part of their definition. Comments delimited with `/*` and `*/`, `//`, `#;`, `;`, and `;;` all behave in their usual way. See Comments.

Comments that begin with the `///` indicator have a special functionality. If you want your IDE to provide typeahead assistance for a macro that is in an include file, then place a `///` comment on the line that immediately precedes its definition; this causes its name to appear in the IDE typeahead popup. (All macros in the current file appear in that popup, without any intervention on your part.) For example, if the following code were referenced through an **#include** directive, then the first macro would appear in typeahead popup and the second would not:

ObjectScript

```
/// A macro that is visible with IDE Typeahead
#define MyAssistMacro 100
 //
 // ...
 //
 // A macro that is not visible with IDE Typeahead
#define MyOtherMacro -100
```

For information on making macros available through include files, see Including Include Files.

18.4 Including Include Files

This section describes how to include include files in your code.

- To include an include file in a class or at the beginning of a routine, use a directive of the form:

  ### ObjectScript

  ```
  #include MacroIncFile
  ```

  where *MacroIncFile* refers to an included file containing macros that is called MacroIncFile.inc. Note that the .inc suffix is not included in the name of the referenced file when it is an argument of **#include**. The **#include** directive is not case-sensitive.

  Note that when a class includes an include file, any subclass of that class automatically includes the same include file.

  For example, if you have one or more macros in the file MyMacros.inc, you can include them with the following call:

  ### ObjectScript

  ```
  #include MyMacros
  ```

- To include multiple include files in a routine, use multiple directives of the same form. For example:

  ### ObjectScript

  ```
  #include MyMacros
  #include YourMacros
  ```

- To include multiple include files at the beginning of a class definition, the syntax is of the form:

  ```
  include (MyMacros, YourMacros)
  ```

  Note that this `include` syntax does not have a leading pound sign; this syntax cannot be used for `#include`.

See the reference section on #include.

Note that when you compile a class definition, that process normalizes the class definition in various ways such as removing whitespace. One of these normalizations converts the capitalization of the include directive.

The ObjectScript compiler provides a `/defines` qualifier that permits including external macros. For further details refer to the Compiler Qualifiers table in the **$SYSTEM** reference page.

18.5 Where to See Expanded Macros

As noted above, when you compile classes and routines, the system generates INT code (intermediate ObjectScript) code, which you can display and read the INT code, which is a useful way to perform some kinds of troubleshooting.

Note: The preprocessor expands macros before the ObjectScript parser handles any Embedded SQL. The preprocessor supports Embedded SQL in either embedded or deferred compilation mode; the preprocessor does not expand macros within Dynamic SQL.

The ObjectScript parser removes multiple line comments before parsing preprocessor directives. Therefore, any macro preprocessor directive specified within a /* . . . */ multiple line comment is not executed.

Also, the following globals contain MAC code (the original source code). Use **ZWRITE** to display these globals and their subscripts:

- **^rINDEX(routinename,"MAC")** contains the timestamp when the MAC code was last saved after being modified, and the character count for this MAC code file. The character count including comments and blank lines. The timestamp when the MAC code was last saved, when it was compiled, and information about #include files used are recorded in the ^ROUTINE global for the INT code. For further details about INT code, refer to the ZLOAD command.
- **^rMAC(routinename)** contains a subscript node for each line of code in the MAC routine, as well as `^rMAC(routinename,0,0)` containing the line count, `^rMAC(routinename,0)` containing the timestamp when it was last saved, and `^rMAC(routinename,0,"SIZE")` containing the character count.
- **^rMACSAVE(routinename)** contains the history of the MAC routine. It contains the same information as **^rMAC(routinename)** for the past five saved versions of the MAC routine. It does not contain information about the current MAC version.

18.6 See Also

- #define
- #include
- Preprocessor Directives Reference
- System Macros

19
Embedded SQL

You can embed SQL within ObjectScript.

19.1 Embedded SQL

Embedded SQL allows you to include SQL code within an ObjectScript program. The syntax is `&sql( )`. For example:

ObjectScript

```
&sql( SELECT Name INTO :n FROM Sample.Person )
WRITE "name is: ",n
```

Embedded SQL is not compiled when the routine that contains it is compiled. Instead, compilation of Embedded SQL occurs upon the first execution of the SQL code (runtime).

For further details, see Using Embedded SQL.

19.2 Other Forms of Queries

You can include SQL queries in other ways within ObjectScript, by using the API provided by the %SQL classes. See Using Dynamic SQL.

20 Locking and Concurrency Control

An important feature of any multi-process system is concurrency control, the ability to prevent different processes from changing a specific element of data at the same time, resulting in corruption. Consequently, InterSystems IRIS® data platform provides a lock management system. This page provides an overview.

20.1 Introduction

The basic locking mechanism is the **LOCK** command. The purpose of this command is to delay activity in one process until another process has signaled that it is OK to proceed.

In InterSystems IRIS, a lock does not, by itself, prevent activity. Locking works only by convention: it requires that mutually competing processes all implement locking with the same lock names. For example, the following describes a common scenario:

1. Process A issues the **LOCK** command, and InterSystems IRIS creates a lock (by default, an exclusive lock).

   Typically, process A then makes changes to nodes in a global. The details are application-specific.

2. Process B issues the **LOCK** command with the same lock name. Because there is an existing exclusive lock, process B pauses. Specifically, the **LOCK** command does not return, and no successive lines of code can be executed.

3. When the process A releases the lock, the **LOCK** command in process B finally returns and process B continues.

   Typically, process B then makes changes to nodes in the same global.

20.2 Lock Names

One of the arguments for the **LOCK** command is the lock name. Lock names are arbitrary, but by universal convention, programmers use lock names that are identical to the names of the item to be locked. Usually the item to be locked is a global or a node of a global. Thus lock names usually look like names of global names or names of nodes of globals. (This page discusses only lock names that start with carets, because those are the most common; for details on locks with name that do not start with carets, see LOCK.)

Formally, lock names follow the same naming conventions as local variables and global variables, as described in Variables. Like variables, lock names are case-sensitive and can have subscripts. Do not use process-private global names as lock names (you would not need such a lock anyway because by definition only one process can access such a global).

Tip: Because locking works by convention and because lock names are arbitrary, it is not necessary to define a given variable *before* creating a lock with the same name.

The form of the lock name has an effect on performance, because of how InterSystems IRIS allocates and manages memory. Locking is optimized for lock names that use subscripts. An example is `^name("ABC")`.

In contrast, InterSystems IRIS is not optimized for lock names such as `^nameABC` or `^nameDEF`. Non-subscripted lock names can also cause performance problems related to ECP.

20.3 The Lock Table

InterSystems IRIS maintains a system-wide, in-memory table that records all current locks and the processes that have own them. This table — the lock table — is accessible via the Management Portal, where you can view the locks and (in rare cases, if needed) remove them. Note that any given process can own multiple locks, with different lock names (or even multiple locks with the same lock name).

When a process ends, the system automatically releases all locks that the process owns. Thus it is not generally necessary to remove locks via the Management Portal, except in the case of an application error.

The lock table cannot exceed a fixed size, which you can specify using the locksiz setting. For information, see Monitoring Locks. Consequently, it is possible for the lock table to fill up, such that no further locks are possible. If this occurs, InterSystems IRIS writes the following message to the messages.log file:

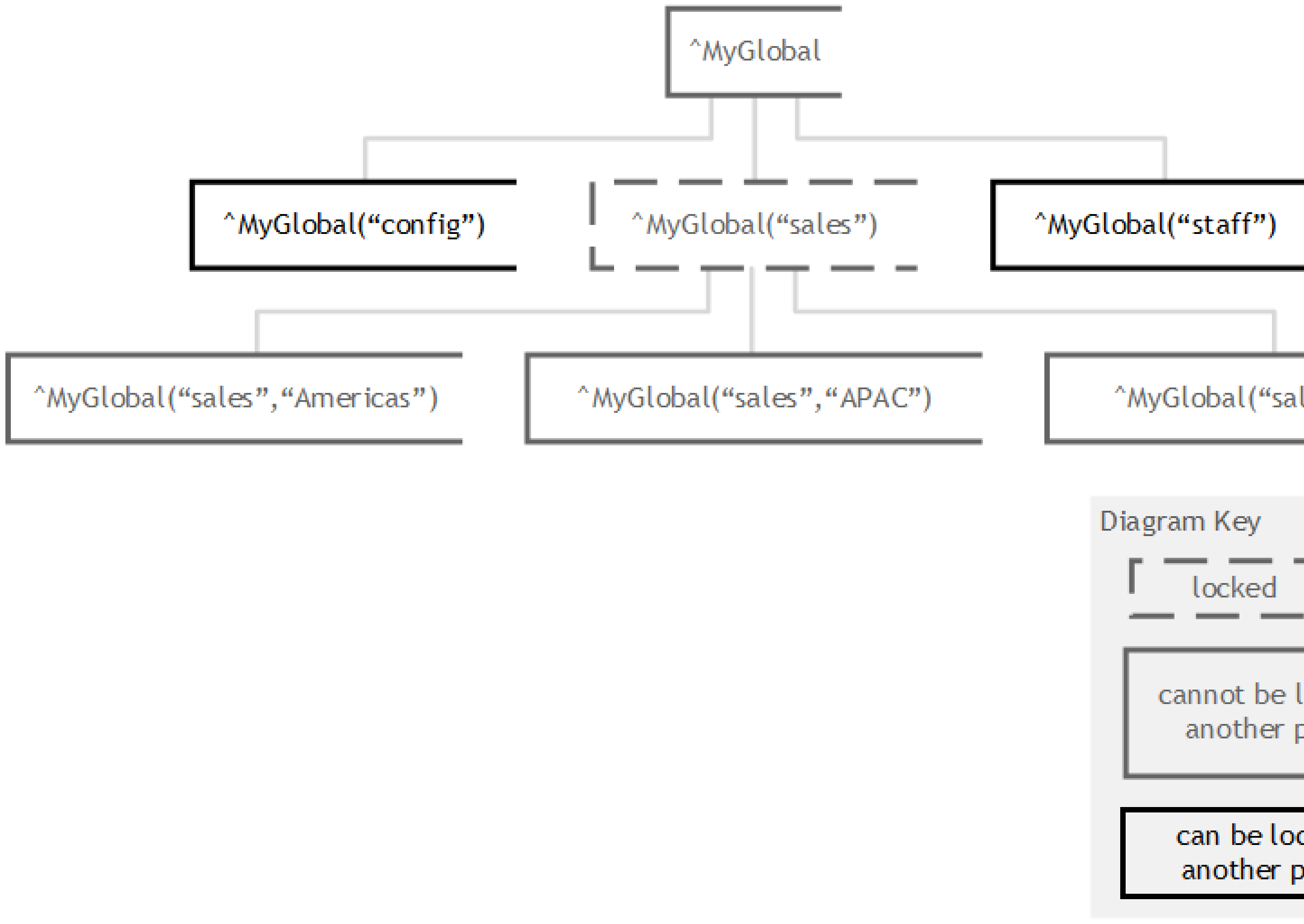
```
LOCK TABLE FULL
```

Filling the lock table is *not* generally considered to be an application error; InterSystems IRIS also provides a lock queue, and processes wait until there is space to add their locks to the lock table. (However, deadlock *is* considered an application programming error. See Avoiding Deadlock.)

20.4 Locks and Arrays

When you lock an array, you can lock either the entire array or one or more nodes in the array. When you lock an array node, other processes are blocked from locking any node that is subordinate to that node. Other processes are also blocked from locking the direct ancestors of the locked node.

The following figure shows an example:



Implicit locks are not included in the lock table and thus do not affect the size of the lock table.

The InterSystems IRIS lock queuing algorithm queues all locks for the same lock name in the order received, even when there is no direct resource contention. For an example and details, see Queuing of Array Node Locks.

20.5 Using the LOCK Command

This section discusses how to use the LOCK command to add and remove locks.

20.5.1 Adding an Incremental Lock

To add a lock, use the **LOCK** command as follows:

```
LOCK +lockname
```

Where *lockname* is the literal lock name. The plus sign (+) creates an incremental lock, which is the common scenario; see Creating Simple Locks for a less common alternative.

This command does the following:

1. Attempts to add the given lock to the lock table. That is, this entry is added to the lock queue.

2. Pauses execution until the lock can be acquired.

There are different types of locks, which behave differently. To add a lock of a non-default lock type, use the following variation:

```
LOCK +lockname#locktype
```

Where *locktype* is a string of lock type codes enclosed in double quotes; see Lock Types.

Note that a given process can add multiple incremental locks with the same name; these locks can be of different types or can all be the same type.

20.5.2 Adding an Incremental Lock with a Timeout

If used incorrectly, incremental locks can result in an undesirable situation known as *deadlock*, discussed later in Avoiding Deadlock. One way to avoid deadlock is to specify a timeout period when you create a lock. To do so, use the **LOCK** command as follows:

```
LOCK +lockname#locktype :timeout
```

Where *timeout* is the timeout period in seconds. The space before the colon is optional. If you specify *timeout* as 0, InterSystems IRIS makes one attempt to add the lock (but see the note, below).

This command does the following:

1. Attempts to add the given lock to the lock table. That is, this entry is added to the lock queue.
2. Pauses execution until the lock can be acquired or until the timeout period ends, whichever comes first.
3. Sets the value of the **$TEST** special variable. If the lock is acquired, InterSystems IRIS sets **$TEST** equal to 1. Otherwise, InterSystems IRIS sets **$TEST** equal to 0.

This means that if you use the timeout argument, your code should next check the value of the **$TEST** special variable and use the value to choose whether to proceed. The following shows an example:

ObjectScript

```
 Lock +^ROUTINE(routinename):0
 If '$TEST {  Return $$$ERROR("Cannot lock the routine: ",routinename)}
```

20.5.2.1 A Note on the Zero Timeout

As noted above, if you specify *timeout* as 0, InterSystems IRIS makes one attempt to add the lock. However, if you try to take a lock on a parent node using a zero timeout, and you already have a lock on a child node, the zero timeout is ignored and there is an internal 1 second timeout, which is used instead.

20.5.3 Removing a Lock

To remove a lock of the default type, use the **LOCK** command as follows:

```
LOCK -lockname
```

If the process that executes this command owns a lock (of the default type) with the given name, this command removes that lock. Or if the process owns more than one lock (of the default type), this command removes one of them.

Or to remove a lock of another type:

```
LOCK -lockname#locktype
```

Where *locktype* is a string of lock type codes; see Lock Types. The lock type codes do *not* have to be in the same order as when the lock was created.

20.5.4 Other Basic Variations of the LOCK Command

For completeness, this section discusses the other basic variations of the **LOCK** command: using it to create simple locks and using it to remove all locks. These variations are uncommon in practice.

20.5.4.1 Creating Simple Locks

For the **LOCK** command, if you omit the **+** operator, the **LOCK** command first removes all existing locks held by this process and then attempts to add the new lock. In this case, the lock is called a *simple lock* rather than an incremental lock. It is possible for a process to own multiple simple locks, *if* that process creates them all at the same time with syntax like the following:

```
 LOCK (^MyVar1,^MyVar2,^MyVar3)
```

Simple locks are not common in practice, because it is usually necessary to hold multiple locks and to acquire them at different steps in your code. Thus it is more practical to use incremental locks.

However, if simple locks are appropriate for you, note that you can specify the *locktype* and *timeout* arguments when you create a simple lock. Also, to remove a simple lock, you can use the **LOCK** command with a minus sign (`-`).

20.5.4.2 Removing All Locks

To remove all locks held by the current process, use the **LOCK** command with no arguments. In practice, it is not common to use the command this way, for two reasons:

- It is best to release specific locks as soon as possible.
- When the process ends, all its locks are automatically released.

20.6 Lock Types

The *locktype* argument specifies the type of lock to add or remove. When adding a lock, include this argument as follows:

```
LOCK +lockname#locktype
```

Or when removing a lock:

```
LOCK -lockname#locktype
```

In either case, *locktype* is one or more lock type codes (in any order) enclosed in double quotes. Note that if you specify the *locktype* argument, you must include a pound character (#) to separate the lock name from the lock type.

There are four lock type codes, as follows. Note that these are not case-sensitive.

- `S` — Adds a shared lock. See Exclusive and Shared Locks.
- `E` — Adds an escalating lock. See Non-Escalating and Escalating Locks.
- `I` — Adds a lock with immediate unlock.
- `D` — Adds a lock with deferred unlock.

> The lock type codes `D` and `I` have special behavior in transactions. For details, see LOCK. You cannot use these two lock type codes at the same time for the same lock name.

The next sections discuss the most common variations, and the last subsection summarizes all the lock types.

20.6.1 Exclusive and Shared Locks

Any lock is either *exclusive* (the default) or *shared*. These types have the following significance:

- While one process owns an exclusive lock (with a given lock name), no other process can acquire any lock with that lock name.
- While one process owns a shared lock (with a given lock name), other processes can acquire shared locks with that lock name, but no other process can acquire an exclusive lock with that lock name.

The typical purpose of an exclusive lock is to indicate that you intend to modify a value and that other processes should not attempt to read or modify that value. The typical purpose of a shared lock is to indicate that you intend to read a value and that other processes should not attempt to modify that value; they can, however, read the value. Also see Practical Uses for Locks.

20.6.2 Non-Escalating and Escalating Locks

Any lock is also either *non-escalating* (the default) or *escalating*. The purpose of escalating locks is to make it easier to manage large numbers of locks, which consume memory and which increase the chance of filling the lock table.

You use escalating locks when you lock multiple nodes of the same array. For escalating locks, if a given process has created more than a specific number (by default, 1000) of locks on parallel nodes of a given array, InterSystems IRIS replaces the individual lock names and replaces them with a new lock that contains the lock count. (In contrast, InterSystems IRIS never does this for non-escalating locks.) For an example and additional details, see Escalating Locks.

Note: You can create escalating locks only for lock names that include subscripts. If you attempt to create an escalating lock with a lock name that has no subscript, InterSystems IRIS issues a <COMMAND> error.

20.6.3 Summary of Lock Types

The following table lists all the possible lock types with their descriptions:

	Exclusive Locks	**Shared Locks (`#"S"` locks)**
Non-escalating Locks	• *locktype omitted* — Default lock type • `#"I"` — Exclusive lock with immediate unlock • `#"D"` — Exclusive lock with deferred unlock	• `#"S"` — Shared lock • `#"SI"` — Shared lock with immediate unlock • `#"SD"` — Shared lock with deferred unlock
Escalating Locks (`#"E"` locks)	• `#"E"` — Exclusive escalating lock • `#"EI"` — Exclusive escalating lock with immediate unlock • `#"ED"` — Exclusive escalating lock with deferred unlock	• `#"SE"` — Shared escalating lock • `#"SEI"` — Shared escalating lock with immediate unlock • `#"SED"` — Shared escalating lock with deferred unlock

For any lock type that uses multiple lock codes, the lock codes can be in any order. For example, the lock type `#"SI"` is equivalent to `#"IS"`.

For details on immediate unlock and deferred unlock, see LOCK. You cannot use these two lock type codes at the same time for the same lock name.

20.7 Escalating Locks

You use escalating locks to manage large numbers of locks. They are relevant when you lock nodes of an array, specifically when you lock multiple nodes at the same subscript level.

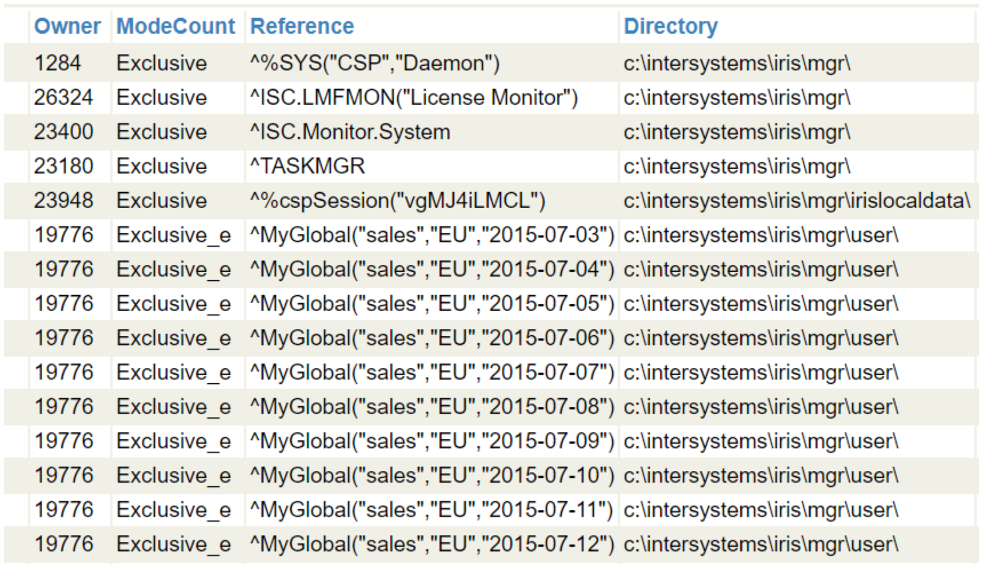
When a given process has created more than a specific number (by default, 1000) of escalating locks at a given subscript level in the same array, InterSystems IRIS removes all the individual lock names and replaces them with a new lock. The new lock is at the parent level, which means that this entire branch of the array is implicitly locked. The example (shown next) demonstrates this.

Your application should release locks for specific child nodes as soon as it is suitable to do so (exactly as with non-escalating locks). As you release locks, InterSystems IRIS decrements the corresponding lock count. When your application removes enough locks, InterSystems IRIS removes the lock on the parent node. The second subsection shows an example.

For information on specifying the lock threshold (which by default is 1000), see LockThreshold.

20.7.1 Lock Escalation Example

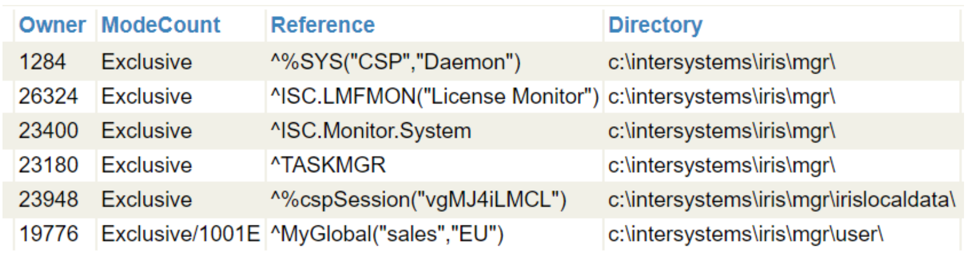
Suppose that you have 1000 locks of the form `^MyGlobal("sales","EU",salesdate)` where *salesdate* represents dates. The lock table might look like this:



Owner	ModeCount	Reference	Directory
1284	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\iris\mgr\
26324	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\iris\mgr\
23400	Exclusive	^ISC.Monitor.System	c:\intersystems\iris\mgr\
23180	Exclusive	^TASKMGR	c:\intersystems\iris\mgr\
23948	Exclusive	^%cspSession("vgMJ4iLMCL")	c:\intersystems\iris\mgr\irislocaldata\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-03")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-04")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-05")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-06")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-07")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-08")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-09")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-10")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-11")	c:\intersystems\iris\mgr\user\
19776	Exclusive_e	^MyGlobal("sales","EU","2015-07-12")	c:\intersystems\iris\mgr\user\

Notice the entries for **Owner** 19776 (this is the process that owns the lock). The **ModeCount** column indicates that these are exclusive, escalating locks.

When the same process attempts to create another lock of the same form, InterSystems IRIS escalates them. It removes these locks and replaces them with a single lock of the name `^MyGlobal("sales","EU")`. Now the lock table might look like this:



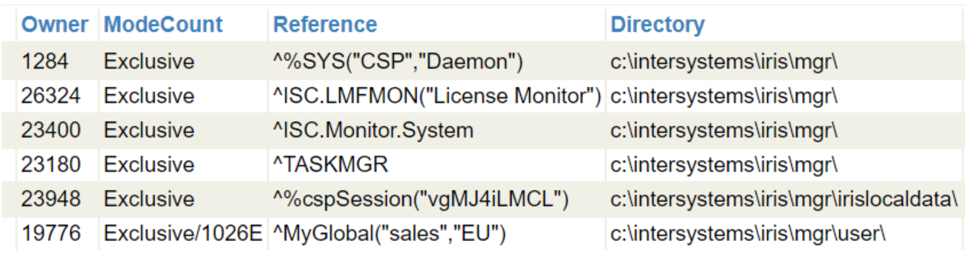
Owner	ModeCount	Reference	Directory
1284	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\iris\mgr\
26324	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\iris\mgr\
23400	Exclusive	^ISC.Monitor.System	c:\intersystems\iris\mgr\
23180	Exclusive	^TASKMGR	c:\intersystems\iris\mgr\
23948	Exclusive	^%cspSession("vgMJ4iLMCL")	c:\intersystems\iris\mgr\irislocaldata\
19776	Exclusive/1001E	^MyGlobal("sales","EU")	c:\intersystems\iris\mgr\user\

The **ModeCount** column indicates that this is a shared, escalating lock and that its count is 1001.

Note the following key points:

- All child nodes of `^MyGlobal("sales","EU")` are now implicitly locked, following the basic rules for array locking.
- The lock table no longer contains information about which child nodes of `^MyGlobal("sales","EU")` were specifically locked. This has important implications when you remove locks; see the next subsection.

When the same process adds more lock names of the form `^MyGlobal("sales","EU",salesdate)`, the lock table increments the lock count for the lock name `^MyGlobal("sales","EU")`. The lock table might then look like this:

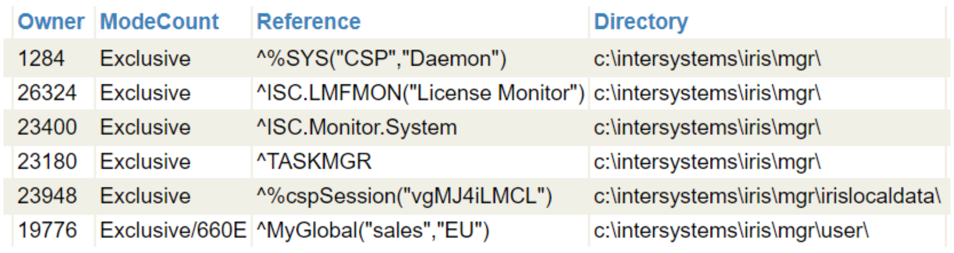


Owner	ModeCount	Reference	Directory
1284	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\iris\mgr\
26324	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\iris\mgr\
23400	Exclusive	^ISC.Monitor.System	c:\intersystems\iris\mgr\
23180	Exclusive	^TASKMGR	c:\intersystems\iris\mgr\
23948	Exclusive	^%cspSession("vgMJ4iLMCL")	c:\intersystems\iris\mgr\irislocaldata\
19776	Exclusive/1026E	^MyGlobal("sales","EU")	c:\intersystems\iris\mgr\user\

The **ModeCount** column indicates that the lock count for this lock is now 1026.

20.7.2 Removing Escalating Locks

In exactly the same way as with non-escalating locks, your application should release locks for specific child nodes as soon as possible. As you do so, InterSystems IRIS decrements the lock count for the escalated lock. For example, suppose that your code removes the locks for `^MyGlobal("sales","EU",salesdate)` where *salesdate* corresponds to any date in 2011 — thus removing 365 locks. The lock table now looks like this:



Owner	ModeCount	Reference	Directory
1284	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\iris\mgr\
26324	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\iris\mgr\
23400	Exclusive	^ISC.Monitor.System	c:\intersystems\iris\mgr\
23180	Exclusive	^TASKMGR	c:\intersystems\iris\mgr\
23948	Exclusive	^%cspSession("vgMJ4iLMCL")	c:\intersystems\iris\mgr\irislocaldata\
19776	Exclusive/660E	^MyGlobal("sales","EU")	c:\intersystems\iris\mgr\user\

Notice that even though the number of locks is now below the threshold (1000), the lock table does not contain individual entries for the locks for `^MyGlobal("sales","EU",salesdate)`.

The node `^MyGlobal("sales")` remains explicitly locked until the process removes 661 more locks of the form `^MyGlobal("sales","EU",salesdate)`.

Important: There is a subtle point to consider, related to the preceding discussion. It is possible for an application to “release” locks on array nodes that were never locked in the first place, thus resulting in an inaccurate lock count for the escalated lock — and possibly releasing the escalated lock before it is desirable to do so.

For example, suppose that the process locked nodes in `^MyGlobal("sales","EU",salesdate)` for the years 2010 through the present. This would create more than 1000 locks and this lock would be escalated, as planned. Suppose that a bug in the application removes locks for the nodes for the year 1970. InterSystems IRIS would permit this action, even though those nodes were not previously locked, and InterSystems IRIS would decrement the lock count by 365. The resulting lock count would not be an accurate count of the desired locks. If the application then removed locks for other years, the escalated lock could potentially be removed unexpectedly early.

20.8 Locks, Globals, and Namespaces

Locks are typically used to control access to globals. Because a global can be accessed from multiple namespaces, InterSystems IRIS provides automatic cross-namespace support for its locking mechanism. The behavior is automatic and needs no intervention, but is described here for reference. There are several scenarios to consider:

- Any namespace has a default database which contains data for persistent classes and any additional globals; this is the *globals database* for this namespace. When you access data (in any manner), InterSystems IRIS retrieves it from this database unless other considerations apply. A given database can be the globals database for more than one namespace. See Scenario 1.
- A namespace can include mappings that provide access to globals stored in other databases. See Scenario 2.
- A namespace can include subscript level global mappings that provide access to globals partly stored in other databases. See Scenario 3.
- Code running in one namespace can use an extended reference to access a global not otherwise available in this namespace. See Scenario 4.

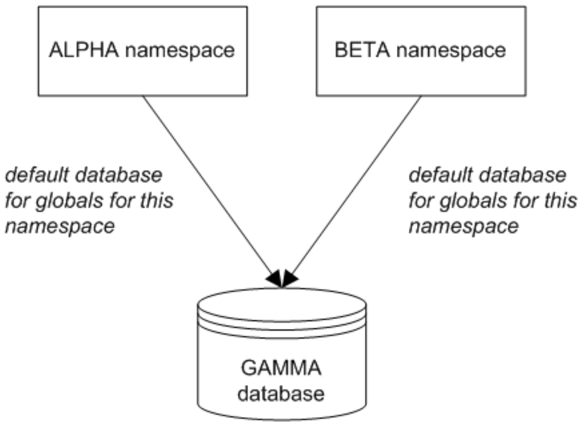
Although lock names are intrinsically arbitrary, when you use a lock name that starts with a caret (`^`), InterSystems IRIS provides special behavior appropriate for these scenarios. The following subsections give the details. For simplicity, only exclusive locks are discussed; the logic is similar for shared locks.

20.8.1 Scenario 1: Multiple Namespaces with the Same Globals Database

As noted earlier, while process A owns an exclusive lock with a given lock name, no other process can acquire any lock with the same lock name.

If the lock name starts with a caret, this rule applies to *all* namespaces that use the same globals database.

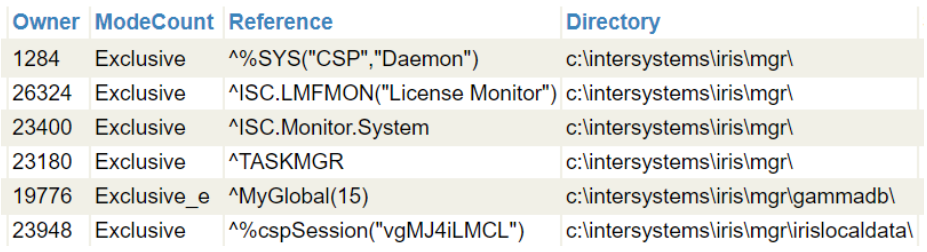
For example, suppose that the namespaces `ALPHA` and `BETA` are both configured to use database `GAMMA` as their globals database. The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
default database for globals for this namespace
GAMMA database

Then consider the following scenario:

1. In namespace ALPHA, process A acquires an exclusive lock with the name `^MyGlobal(15)`.
2. In namespace BETA, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

In this scenario, the lock table contains only the entry for the lock owned by Process A. If you examine the lock table, you will notice that it indicates the *database* to which this lock applies; see the **Directory** column. For example:

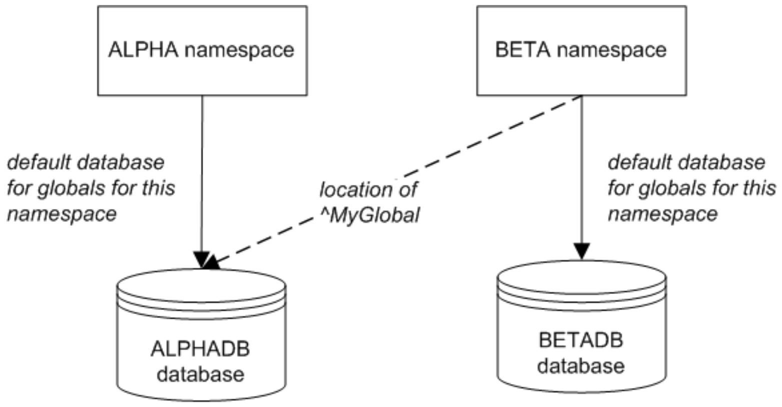


Owner	ModeCount	Reference	Directory
1284	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\iris\mgr\
26324	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\iris\mgr\
23400	Exclusive	^ISC.Monitor.System	c:\intersystems\iris\mgr\
23180	Exclusive	^TASKMGR	c:\intersystems\iris\mgr\
19776	Exclusive_e	^MyGlobal(15)	c:\intersystems\iris\mgr\gammadb\
23948	Exclusive	^%cspSession("vgMJ4iLMCL")	c:\intersystems\iris\mgr\irislocaldata\

20.8.2 Scenario 2: Namespace Uses a Mapped Global

If one or more namespaces include global mappings, the system automatically enforces the lock mechanism across the applicable namespaces. InterSystems IRIS automatically creates additional lock table entries when locks are acquired in the non-default namespace.

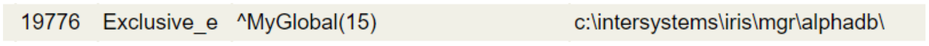
For example, suppose that namespace ALPHA is configured to use database ALPHADB as its globals database. Suppose that namespace BETA is configured to use a different database (BETADB) as its globals database. The namespace BETA also includes a global mapping that specifies that `^MyGlobal` is stored in the ALPHADB database. The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
location of ^MyGlobal
default database for globals for this namespace
ALPHADB database
BETADB database

Then consider the following scenario:

1. In namespace `ALPHA`, process A acquires an exclusive lock with the name `^MyGlobal(15)`.

   As with the previous scenario, the lock table contains only the entry for the lock owned by Process A. This lock applies to the `ALPHADB` database:

   ```
   19776   Exclusive_e   ^MyGlobal(15)                    c:\intersystems\iris\mgr\alphadb\
   ```

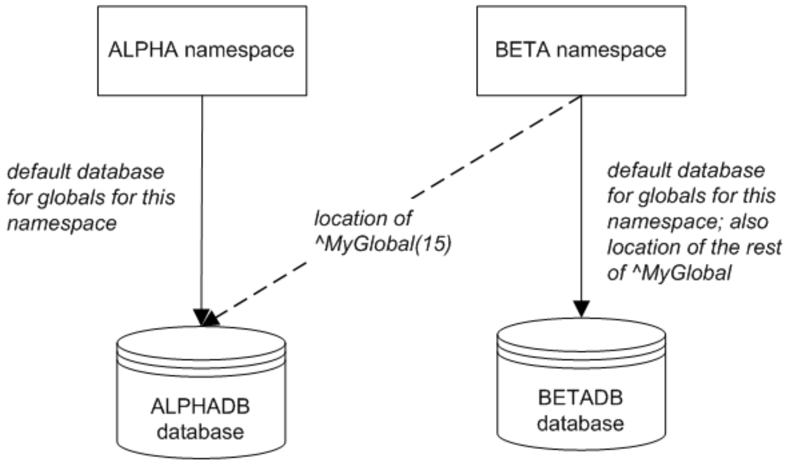
2. In namespace `BETA`, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

20.8.3 Scenario 3: Namespace Uses a Mapped Global Subscript

If one or more namespaces include global mappings that use subscript level mappings, the system automatically enforces the lock mechanism across the applicable namespaces. In this case, InterSystems IRIS also automatically creates additional lock table entries when locks are acquired in a non-default namespace.

For example, suppose that namespace `ALPHA` is configured to use the database `ALPHADB` as its globals database. Namespace `BETA` uses the `BETADB` database as its globals database.

Also suppose that the namespace `BETA` also includes a subscript-level global mapping so that `^MyGlobal(15)` is stored in the `ALPHADB` database (while the rest of this global is stored in the namespace's default location). The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
location of ^MyGlobal(15)
default database for globals for this namespace; also location of the rest of ^MyGlobal
ALPHADB database
BETADB database

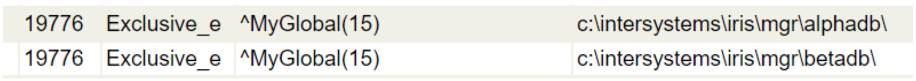
Then consider the following scenario:

1. In namespace `ALPHA`, process A acquires an exclusive lock with the name `^MyGlobal(15)`.

   As with the previous scenario, the lock table contains only the entry for the lock owned by Process A. This lock applies to the `ALPHADB` database (`c:\InterSystems\IRIS\mgr\alphadb`, for example).

2. In namespace `BETA`, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

When a non-default namespace acquires a lock, the overall behavior is the same, but InterSystems IRIS handles the details slightly differently. Suppose that in namespace `BETA`, a process acquires a lock with the name `^MyGlobal(15)`. In this case, the lock table contains two entries, one for the `ALPHADB` database and one for the `BETADB` database. Both locks are owned by the process in namespace `BETA`.



19776	Exclusive_e	^MyGlobal(15)	c:\intersystems\iris\mgr\alphadb\
19776	Exclusive_e	^MyGlobal(15)	c:\intersystems\iris\mgr\betadb\

When this process releases the lock name `^MyGlobal(15)`, the system automatically removes both locks.

20.8.4 Scenario 4: Extended Global References

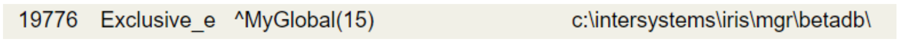
Code running in one namespace can use an extended reference to access a global not otherwise available in this namespace. In this case, InterSystems IRIS adds an entry to the lock table that affects the relevant database. The lock is owned by the process that created it. For example, consider the following scenario. For simplicity, there are no global mappings in this scenario.

1. Process A is running in the `ALPHA` namespace, and this process uses the following command to acquire a lock on a global that is available in the `BETA` namespace:

   ObjectScript

   ```
   lock ^["beta"]MyGlobal(15)
   ```

2. Now the lock table includes the following entry:

19776	Exclusive_e	^MyGlobal(15)	c:\intersystems\iris\mgr\betadb\

   Note that this shows only the global name (rather than the reference used to access it). Also, in this scenario, `BETADB` is the default database for the `BETA` namespace.

3. In namespace `BETA`, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

A process-private global is technically a kind of extended reference, but InterSystems IRIS does not support using a process-private global names as lock names; you would not need such a lock anyway because by definition only one process can access such a global.

20.9 Avoiding Deadlock

Incremental locking is potentially dangerous because it can lead to a situation known as *deadlock*. This situation occurs when two processes each assert an incremental lock on a variable already locked by the other process. Because the attempted locks are incremental, the existing locks are not released. As a result, each process hangs while waiting for the other process to release the existing lock.

As an example:

1. Process A issues this command: `lock + ^MyGlobal(15)`
2. Process B issues this command: `lock + ^MyOtherGlobal(15)`
3. Process A issues this command: `lock + ^MyOtherGlobal(15)`

   This **LOCK** command does not return; the process is blocked until process B releases this lock.
4. Process B issues this command: `lock + ^MyGlobal(15)`

   This **LOCK** command does not return; the process is blocked until process A releases this lock. Process A, however, is blocked and cannot release the lock. Now these processes are both waiting for each other.

There are several ways to prevent deadlocks:

- Always include the timeout argument.
- Follow a strict protocol for the order in which you issue incremental **LOCK** commands. Deadlocks cannot occur as long as all processes follow the same order for lock names. A simple protocol is to add locks in collating sequence order.
- Use simple locking rather than incremental locking; that is, do not use the **+** operator. As noted earlier, with simple locking, the **LOCK** command first releases all previous locks held by the process. (In practice, however, simple locking is not often used.)

If a deadlock occurs, you can resolve it by using the Management Portal or the **`^LOCKTAB`** routine. See Monitoring Locks.

20.10 Practical Uses for Locks

This section presents the basic ways in which locks are used in practice.

20.10.1 Controlling Access to Application Data

Locks are used very often to control access to application data, which is stored in globals. Your application might need to read or modify a particular piece or pieces of this data, and your application would create one or more locks before doing so, as follows:

- If your application needs to read one or more global nodes, and you do not want other processes to modify the values during the read operation, create shared locks for those nodes.
- If your application needs to modify one or more global nodes, and you do not want other processes to read these nodes during the modification, create exclusive locks for those nodes.

Then either read or make the modifications as planned. When you are done, remove the locks.

Remember that the locking mechanism works purely by convention. Any other code that would read or modify these nodes must also attempt to acquire locks before performing those operations.

20.10.2 Preventing Simultaneous Activity

Locks are also used to prevent multiple processes from performing the same activity. In this scenario, you also use a global, but the global contains data for the internal purposes of your application, rather than pure application data. As a simple example, suppose that you have a routine (`^NightlyBatch`) that should never be run by more than one process at any given time. This routine could do the following, at a very early stage in its processing:

1. Create an exclusive lock on a specific global node, for example, `^AppStateData("NightlyBatch")`. Specify a timeout for this operation.
2. If the lock is acquired, set nodes in a global to record that the routine has been started (as well as any other relevant information). For example:

   ObjectScript

   ```
    set ^AppStateData("NightlyBatch")=1
    set ^AppStateData("NightlyBatch","user")=$USERNAME
   ```

   Or, if the lock is not acquired within the timeout period, quit with an error message that indicates that this routine has already been started.

Then, at the end of its processing, the same routine would clear the applicable global nodes and release the lock.

The following partial example demonstrates this technique, which is adapted from code that InterSystems IRIS uses internally:

ObjectScript

```
  lock ^AppStateData("NightlyBatch"):0
  if '$TEST {
     write "You cannot run this routine right now."
     write !, "This routine is currently being run by user: "_^AppStateData("NightlyBatch","user")
     quit
  }
  set ^AppStateData("NightlyBatch")=1
  set ^AppStateData("NightlyBatch","user")=$USERNAME
  set ^AppStateData("NightlyBatch","starttime")=$h

  //main routine activity omitted from example

  kill ^AppStateData("NightlyBatch")
  lock -^AppStateData("NightlyBatch")
```

20.11 Locking and Concurrency in SQL and Persistent Classes

When you work with InterSystems SQL or persistent classes, you do not need to directly use the ObjectScript LOCK command because there are alternatives suitable for your use cases. (Internally these alternatives all use LOCK.)

- InterSystems SQL provides commands for working with locks. For details, see the InterSystems SQL Reference. Similarly, the system automatically performs locking on INSERT, UPDATE, and DELETE operations (unless you specify the %NOLOCK keyword).
- The %Persistent class provides a way to control concurrent access to objects, namely, the concurrency argument to **%OpenId()** and other methods of this class. All persistent objects inherit these methods. See Object Concurrency.

  The %Persistent class also provides the methods **%GetLock()**, **%ReleaseLock()**, **%LockId()**, **%UnlockId()**, **%LockExtent()**, and **%UnlockExtent()**. For details, see the class reference for %Persistent.

20.12 See Also

- LOCK command reference
- ^$LOCK (**^$LOCK** is a structured system variable that contains information about locks.)

- Transaction Processing
- Details of Lock Requests and Deadlocks
- Managing the Lock Table
- Monitoring Locks

21 Details of Lock Requests and Deadlocks

This topic provides more detailed information on how lock requests are handled in InterSystems IRIS® data platform, as well as a detailed look at deadlock scenarios.

21.1 Waiting Lock Requests

When a process holds an exclusive lock, it causes a wait condition for any other process that attempts to acquire the same lock, or a lock on a higher level node or lower level node of the held lock. When locking subscripted globals (array nodes) it is important to make the distinction between what you lock, and what other processes can lock:

- *What you lock*: you only have an explicit lock on the node you specify, not its higher or lower level nodes. For example, if you lock `^student(1,2)` you only have an explicit lock on `^student(1,2)`. You cannot release this node by releasing a higher level node (such as `^student(1)`) because you don't have an explicit lock on that node. You can, of course, explicitly lock higher or lower nodes in any sequence.
- *What they can lock*: the node that you lock bars other processes from locking that exact node or a higher or lower level node (a parent or child of that node). They cannot lock the parent `^student(1)` because to do so would also implicitly lock the child `^student(1,2)`, which your process has already explicitly locked. They cannot lock the child `^student(1,2,3)` because your process has locked the parent `^student(1,2)`. These other processes wait on the lock queue in the order specified. They are listed in the lock table as waiting on the highest level node specified ahead of them in the queue. This may be a locked node, or a node waiting to be locked.

For example:

1. Process A locks `^student(1,2)`.
2. Process B attempts to lock `^student(1)`, but is barred. This is because if Process B locked `^student(1)`, it would also (implicitly) lock `^student(1,2)`. But Process A holds a lock on `^student(1,2)`. The lock Table lists it as WaitExclusiveParent `^student(1,2)`.
3. Process C attempts to lock `^student(1,2,3)`, but is barred. The lock Table lists it as WaitExclusiveParent `^student(1,2)`. Process A holds a lock on `^student(1,2)` and thus an implicit lock on `^student(1,2,3)`. However, because Process C is lower in the queue than Process B, Process C must wait for Process B to lock and then release `^student(1)`.
4. Process A locks `^student(1,2,3)`. The waiting locks remain unchanged.
5. Process A locks `^student(1)`. The waiting locks change:
    - Process B is listed as WaitExclusiveExact `^student(1)`. Process B is waiting to lock the exact lock (`^student(1)`) that Process A holds.

- Process C is listed as WaitExclusiveChild `^student(1)`. Process C is lower in the queue than Process B, so it is waiting for Process B to lock and release its requested lock. Then Process C will be able to lock the child of the Process B lock. Process B, in turn, is waiting for Process A to release `^student(1)`.

6. Process A unlocks `^student(1)`. The waiting locks change back to WaitExclusiveParent `^student(1,2)`. (Same conditions as steps 2 and 3.)
7. Process A unlocks `^student(1,2)`. The waiting locks change to WaitExclusiveParent `^student(1,2,3)`. Process B is waiting to lock `^student(1)`, the parent of the current Process A lock `^student(1,2,3)`. Process C is waiting for Process B to lock then unlock `^student(1)`, the parent of the `^student(1,2,3)` lock requested by Process C.
8. Process A unlocks `^student(1,2,3)`. Process B locks `^student(1)`. Process C is now barred by Process B. Process C is listed as WaitExclusiveChild `^student(1)`. Process C is waiting to lock `^student(1,2,3)`, the child of the current Process B lock.

21.2 Queuing of Array Node Lock Requests

The basic queuing algorithm for array locks is to queue lock requests for the same resource strictly in the order received, even when there is no direct resource contention. This is illustrated in the following example, in which three locks on the same global array are requested by three different processes in the sequence shown:

```
Process A: LOCK ^x(1,1)
Process B: LOCK ^x(1)
Process C: LOCK ^x(1,2)
```

The status of these requests is as follows:

- Process A holds a lock on `^x(1,1)`.
- Process B cannot lock `^x(1)` until Process A to releases its lock on `^x(1,1)`.
- Process C is also blocked, but not by Process A’s lock; rather, it is the fact that Process B is waiting to explicitly lock `^x(1)`, and thus implicitly lock `^x(1,2)`, that blocks Process C.

This approach is designed to speed the next job in the sequence after the one holding the lock. Allowing Process C to jump Process B in the queue would speed Process C, but could unacceptably delay Process B, especially if there are many jobs like Process C.

The exception to the general rule that requests are processed in the order received is that a process holding a lock on a parent node is immediately granted any requested lock on a child of that node. For example, consider the following extension of the previous example:

```
Process A: LOCK ^x(1,1)
Process B: LOCK ^x(1)
Process C: LOCK ^x(1,2)
Process A: LOCK ^x(1,2)
```

In this case, Process A is immediately granted the requested lock on `^x(1,2)`, ahead of both Process B and Process C, because it already holds a lock on `^x(1,1)`.

Note: This process queuing algorithm applies to all subscripted lock requests. However, the release of a nonsubscripted lock, such as `LOCK ^x`, when there are both nonsubscripted (`LOCK +^x`) and subscripted (`LOCK +^x(1,1)`) requests waiting is a special case, in which the lock request granted is unpredictable and may not follow process queuing.

21.3 ECP Local and Remote Lock Requests

When releasing a lock, an ECP client may donate the lock to a local waiter in preference to waiters on other systems in order to improve performance. The number of times this is allowed to happen is limited in order to prevent unacceptable delays for remote lock waiters.

21.4 Avoiding Deadlock

Requesting a (+) exclusive lock when you hold an existing shared lock is potentially dangerous because it can lead to a situation known as "deadlock". This situation occurs when two processes each request an exclusive lock on a lock name already locked as a shared lock by the other process. As a result, each process hangs while waiting for the other process to release the existing shared lock.

The following example shows how this can occur (numbers indicate the sequence of operations):

Process A	Process B
1. `LOCK ^a(1)#"S"` Process A acquires shared lock.	
	2. `LOCK ^a(1)#"S"` Process B acquires shared lock.
3. `LOCK +^a(1)` Process A requests exclusive lock and waits for Process B to release its shared lock.	
	4. `LOCK +^a(1)` Process B requests exclusive lock and waits for Process A to release its shared lock. Deadlock occurs.

This is the simplest form of deadlock. Deadlock can also occur when a process is requesting a lock on the parent node or child node of a held lock.

To prevent deadlocks, request the exclusive lock without the plus sign, which unlocks your shared lock. In the following example, both processes release their prior locks when requesting an exclusive lock to avoid deadlock (numbers indicate the sequence of operations). Note which process acquires the exclusive lock:

Process A	Process B
1. `LOCK ^a(1)#"S"` Process A acquires shared lock.	
	2. `LOCK ^a(1)#"S"` Process B acquires shared lock.
3. `LOCK ^a(1)` Process A releases shared lock, requests exclusive lock, and waits for Process B to release its shared lock.	
	4. `LOCK ^a(1)` Process B releases shared lock and requests exclusive lock. Process A immediately acquires its requested shared lock. Process B waits for Process A to release its shared lock.

Another way to avoid deadlocks is to follow a strict protocol for the order in which you issue **LOCK +** and **LOCK -** commands. Deadlocks cannot occur as long as all processes follow the same order. A simple protocol is for all processes to apply and release locks in collating sequence order.

To minimize the impact of a deadlock situation, include the *timeout* argument when using plus sign locks. For example, the **LOCK +^a(1):10** operation times out after 10 seconds.

If a deadlock occurs, you can resolve it by using the Management Portal or the **^LOCKTAB** to remove one of the locks in question. From the Management Portal, open the **Manage Locks** window, and then select the **Remove** option for the deadlocked process.

21.5 See Also

- Locking and Concurrency Control
- Managing the Lock Table

22

Managing the Lock Table

This topic discusses tools for viewing and managing the lock table in InterSystems products. (Also see Monitoring Locks.)

22.1 Available Tools for Managing the Lock Table

You may find it necessary to view locks and (occasionally) remove them. InterSystems provides the following tools for this:

- The **Locks** page of the Management Portal. Here you can view locks and remove locks.
- The ^LOCKTAB utility.
- The %SYS.LockQuery class, which lets you read lock table information.
- The SYS.Lock class, which is available in the %SYS namespace

For information on the latter two classes, see the class reference.

22.2 Viewing Locks in the Management Portal

You can view all of the locks currently held or requested (waiting) system-wide using the Management Portal. From the Management Portal, select **System Operation**, select **Locks**, then select **View Locks**. The **View Locks** window displays a list of locks (and lock requests) in alphabetical order by directory (**Directory**) and within each directory in collation sequence by lock name (**Reference**). Each lock is identified by its process id (**Owner**), displays the user name that the operating system gave to the process when it was created (**OS User Name**), and has a **ModeCount** (lock mode and lock increment count). You may need to use the **Refresh** icon to view the most current list of locks and lock requests. For further details on this interface see Monitoring Locks.

ModeCount can indicate a held lock by a specific **Owner** process on a specific **Reference**. The following are examples of **ModeCount** values for held locks:

ModeCount	Description
Exclusive	An exclusive lock, non-escalating (`LOCK +^a(1)`)
Shared	A shared lock, non-escalating (`LOCK +^a(1)#"S"`)
Exclusive_e	An exclusive escalating lock (`LOCK +^a(1)#"E"`)

ModeCount	Description
Exclusive_E	An exclusive/shared escalated lock, as a result of the number of locks on various nodes of this global exceeding the lock threshold
Shared_e	A shared escalating lock (`LOCK +^a(1)#"SE"`)
Shared_E	A shared escalated lock, as a result of the number of locks on various nodes of this global exceeding the lock threshold
Exclusive->Delock	An exclusive lock in a delock state. The lock has been unlocked, but release of the lock is deferred until the end of the current transaction. This can be caused by either a standard unlock (`LOCK -^a(1)`) or a deferred unlock `LOCK -^a(1)#"D"`).
Exclusive,Shared	Both a shared lock and an exclusive lock (applied in any order). Can also specify escalating locks; for example, Exclusive_e,Shared_e
Exclusive/*n*	An incremented exclusive lock (`LOCK +^a(1)` issued *n* times). If the lock count is 1, no count is shown (but see below). Can also specify an incrementing shared lock; for example, Shared/2.
Exclusive/*n*->Delock	An incremented exclusive lock in a delock state. All of the increments of the lock have been unlocked, but release of the lock is deferred until the end of the current transaction. Within a transaction, unlocks of individual increments release those increments immediately; the lock does not go into a delock state until an unlock is issued when the lock count is 1. This ModeCount value, a incremented lock in a delock state, occurs when all prior locks are unlocked by a single operation, either by an argumentless **LOCK** command or a lock with no lock operation indicator (`LOCK ^xyz(1)`).
Exclusive/1+1e	Two exclusive locks, one non-escalating, one escalating. Increment counts are kept separately on these two types of exclusive locks. Can also specify shared locks; for example, Shared/1+1e.
Exclusive/*n*,Shared/*m*	Both a shared lock and an exclusive lock, both with integer increments.

A held lock **ModeCount** can, of course, represent any combination of shared or exclusive, escalating or non-escalating locks — with or without increments. An Exclusive lock or a Shared lock (escalating or non-escalating) can be in a Delock state.

ModeCount can indicate a process waiting for a lock, such as `WaitExclusiveExact`. The following are **ModeCount** values for waiting lock requests:

ModeCount	Description
WaitSharedExact	Waiting for a shared lock on exactly the same lock, either held or previously-requested: `LOCK +^a(1,2)#"S"` is waiting on lock `^a(1,2)`
WaitExclusiveExact	Waiting for an exclusive lock on exactly the same lock, either held or previously-requested: `LOCK +^a(1,2)` is waiting on lock `^a(1,2)`
WaitSharedParent	Waiting for a shared lock on the parent of a held or previously-requested lock: `LOCK +^a(1)#"S"` is waiting on lock `^a(1,2)`
WaitExclusiveParent	Waiting for an exclusive lock on the parent of a held or previously-requested lock: `LOCK +^a(1)` is waiting on lock `^a(1,2)`
WaitSharedChild	Waiting for a shared lock on the child of a held or previously-requested lock: `LOCK +^a(1,2)#"S"` is waiting on lock `^a(1)`
WaitExclusiveChild	Waiting for an exclusive lock on the child of a held or previously-requested lock: `LOCK +^a(1,2)` is waiting on lock `^a(1)`

ModeCount indicates the lock (or lock request) that is blocking this lock request. This is not necessarily the same as **Reference**, which specifies the currently held lock that is at the head of the lock queue on which this lock request is waiting. **Reference** does not necessarily indicate the requested lock that is immediately blocking this lock request.

ModeCount can indicate other lock status values for a specific **Owner** process on a specific **Reference**. The following are these other **ModeCount** status values:

ModeCount	Description
LockPending	An exclusive lock is pending. This status may occur while the server is in the process of granting the exclusive lock. You cannot delete a lock that is in a lock pending state.
SharePending	A shared lock is pending. This status may occur while the server is in the process of granting the shared lock. You cannot delete a lock that is in a lock pending state.
DelockPending	An unlock is pending. This status may occur while the server is in the process of unlocking a held lock. You cannot delete a lock that is in a lock pending state.
Lost	A lock was lost due to network reset.

Select **Display Owner’s Routine Information** to enable the **Routine** column, which provides the name of the routine that the owner process is executing, prepended with the current line number being executed within that routine.

Select **Show SQL Options**, and then select a namespace from the **Show SQL Table Names for Namespace** list, to enable the **SQL Table Name** column. This column provides the name of the SQL table associated with each process in the selected namespace. If the process is not associated with an SQL table, this column value is empty.

The **View Locks** window cannot be used to remove locks.

22.3 Removing Locks in the Management Portal

Important: Rather than removing a lock, the best practice is to identify and then terminate the process that created the lock. Removing a lock can have a severe impact on the system, depending on the purpose of the lock.

To remove (delete) locks currently held on the system, go to the Management Portal, select **System Operation**, select **Locks**, then select **Manage Locks**. For the desired process (**Owner**) click either **Remove** or **Remove All Locks for Process**.

Removing a lock releases all forms of that lock: all increment levels of the lock, all exclusive, exclusive escalating, and shared versions of the lock. Removing a lock immediately causes the next lock waiting in that lock queue to be applied.

You can also remove locks using the **SYS.Lock.DeleteOneLock()** and **SYS.Lock.DeleteAllLocks()** methods.

Removing a lock requires WRITE permission. Lock removal is logged in the audit database (if enabled); it is not logged in messages.log.

22.4 ^LOCKTAB Utility

You can also view and delete (remove) locks using the **^LOCKTAB** utility in the %SYS namespace.

Important: Rather than removing a lock, the best practice is to identify and then terminate the process that created the lock. Removing a lock can have a severe impact on the system, depending on the purpose of the lock.

You can execute **^LOCKTAB** in either of the following forms:

- **DO ^LOCKTAB**: allows you to view and delete locks. It provides letter code commands for deleting an individual lock, deleting all locks owned by a specified process, or deleting all locks on the system.
- **DO View^LOCKTAB**: allows you to view locks. It does not provide options for deleting locks.

Note that these utility names are case-sensitive.

The following Terminal session example shows how **^LOCKTAB** displays the current locks:

```
%SYS>DO ^LOCKTAB

                                   Node Name: MYCOMPUTER
                   LOCK table entries at 07:22AM  01/13/2018
                 16767056 bytes usable, 16774512 bytes available.

Entry Process     X#    S# Flg   W# Item Locked
   1) 4900         1                ^["^^c:\intersystems\iris\mgr\"]%SYS("CSP","Daemon")
   2) 4856         1                ^["^^c:\intersystems\iris\mgr\"]ISC.LMFMON("License Monitor")
   3) 5016         1                ^["^^c:\intersystems\iris\mgr\"]ISC.Monitor.System
   4) 5024         1                ^["^^c:\intersystems\iris\mgr\"]TASKMGR
   5) 6796         1                ^["^^c:\intersystems\iris\mgr\user\"]a(1)
   6) 6796         1e               ^["^^c:\intersystems\iris\mgr\user\"]a(1,1)
   7) 6796               2        1 ^["^^c:\intersystems\iris\mgr\user\"]b(1)Waiters: 3120(XC)
   8) 3120         2                ^["^^c:\intersystems\iris\mgr\user\"]c(1)
   9) 2024         1     1          ^["^^c:\intersystems\iris\mgr\user\"]d(1)

Command=>
```

In the **^LOCKTAB** display, the X# column lists exclusive locks held, the S# column lists shared locks held. The X# or S# number indicates the lock increment count. An “e” suffix indicates that the lock is defined as escalating. A “D” suffix indicates that the lock is in a delock state; the lock has been unlocked, but is not available to another process until the end of the current transaction. The W# column lists number of waiting lock requests. As shown in the above display, process 6796 holds an incremented shared lock ^b(1). Process 3120 has one lock request waiting this lock. The lock request is for an exclusive (X) lock on a child (C) of ^b(1).

Enter a question mark (?) at the `Command=>` prompt to display the help for this utility. This includes further description of how to read this display and letter code commands to delete locks (if available).

Note: You cannot delete a lock that is in a lock pending state, as indicated by the Flg column value.

Enter Q to exit the **^LOCKTAB** utility.

22.5 See Also

- Locking and Concurrency Control
- Details of Lock Requests and Deadlocks

23
Working with %Status Values

When working with an API that returns %Status values (a *status*), it is best practice to check the status before proceeding, and continue with normal processing only in the case of success. In your own code, you can also return status values (and check them elsewhere as appropriate).

This page discusses status values and how to work with them.

Note: Status checking is not error checking per se. Your code should also use TRY-CATCH processing to trap unexpected, unforeseen errors.

23.1 Basics of Working with Status Values

Methods in many InterSystems IRIS® data platform classes return a %Status (%Library.Status) value to indicate success or error. If the status represents an error or errors, the status also includes information about the errors. For example, the **%Save()** method in %Library.Persistent returns a status. For any such method, be sure to obtain the returned status. Then check the status and then proceed appropriately. There are two possible scenarios:

- In the case of success, the status equals 1.
- In the case of failure, the status is an encoded string containing the error status and one or more error codes and text messages. Status text messages are localized for the language of your locale. InterSystems IRIS provides methods and macros for processing the value so that you can understand the nature of the failure.

The basic tools are as follows:

- To check whether the status represents success or error, use any of the following:
  - The `$$$ISOK` and `$$$ISERR` macros, which are defined in the include file %occStatus.inc. This include file is automatically available in all object classes.
  - The **$SYSTEM.Status.IsOK()** and **$SYSTEM.Status.IsError()** methods.
- To display the error details, use **$SYSTEM.Status.DisplayError()** or **$SYSTEM.OBJ.DisplayError()**. These methods are equivalent to each other. They write output to the current device.
- To obtain a string that contains the error details, use **$SYSTEM.Status.GetErrorText()**.

The special variable *$SYSTEM* is bound to the %SYSTEM package. This means that the methods in the previous list are in the %SYSTEM.Status and %SYSTEM.OBJ classes; see the class reference for details.

23.2 Examples

For example:

```
Set object=##class(Sample.Person).%New()
Set object.Name="Smith,Janie"
Set tSC=object.%Save()
If $$$ISERR(tSC) {
  Do $SYSTEM.Status.DisplayError(tSC)
  Quit
}
```

Here is a partial example that shows use of **$SYSTEM.Status.GetErrorText()**:

```
If $$$ISERR(tSC) {
  // if error, log error message so users can see them
  Do ..LogMsg($System.Status.GetErrorText(tSC))
}
```

Note: Some ObjectScript programmers use the letter `t` as a prefix to indicate a temporary variable, so you might see `tSC` used as a variable name in code samples, meaning “temporary status code.” You are free to use this convention, but there is nothing special about this variable name.

23.3 Variation (%objlasterror)

Some methods, such as **%New()**, do not return a %Status but instead update the **%objlasterror** variable to contain the status. **%New()** either returns an OREF to an instance of the class upon success, or the null string upon failure. You can retrieve the status value for methods of this type by accessing the **%objlasterror** variable, as shown in the following example.

ObjectScript

```
 Set session = ##class(%CSP.Session).%New()
 If session="" {
     Write "session OREF not created",!
     Write "%New error is ",!,$System.Status.GetErrorText(%objlasterror),!
 } Else {
     Write "session OREF is ",session,!
 }
```

For more information, refer to the %SYSTEM.Status class.

23.4 Multiple Errors Reported in a Status Value

If a status value represents multiple errors, the previous techniques give you information about only the latest. %SYSTEM.Status provides methods you can use to retrieve individual errors: **GetOneErrorText()** and **GetOneStatusText()**. For example:

ObjectScript

```
CreateCustomErrors
  SET st1 = $System.Status.Error(83,"my unique error")
  SET st2 = $System.Status.Error(5001,"my unique error")
  SET allstatus = $System.Status.AppendStatus(st1,st2)
DisplayErrors
  WRITE "All together:",!
  WRITE $System.Status.GetErrorText(allstatus),!!
  WRITE "One by one",!
  WRITE "First error format:",!
  WRITE $System.Status.GetOneStatusText(allstatus,1),!
  WRITE "Second error format:",!
  WRITE $System.Status.GetOneStatusText(allstatus,2),!
```

Another option is **$SYSTEM.Status.DecomposeStatus()**, which returns an array of the error details (by reference, as the second argument). For example:

```
 Do $SYSTEM.Status.DecomposeStatus(tSC,.errorlist)
 //then examine the errorlist variable
```

The variable *errorlist* is a multidimensional array that contains the error information. The following shows a partial example with some artificial line breaks for readability:

```
ZWRITE errorlist
errorlist=2
errorlist(1)="ERROR #5659: Property 'Sample.Person::SSN(1@Sample.Person,ID=)' required"
errorlist(1,"caller")="%ValidateObject+9^Sample.Person.1"
errorlist(1,"code")=5659
errorlist(1,"dcode")=5659
errorlist(1,"domain")="%ObjectErrors"
errorlist(1,"namespace")="SAMPLES"
errorlist(1,"param")=1
errorlist(1,"param",1)="Sample.Person::SSN(1@Sample.Person,ID=)"
...
errorlist(2)="ERROR #7209: Datatype value '' does not match
PATTERN '3N1""-""2N1""-""4N'"_$c(13,10)_"  >
ERROR #5802: Datatype validation failed on property 'Sample.Person:SSN',
with value equal to """""
errorlist(2,"caller")="zSSNIsValid+1^Sample.Person.1"
errorlist(2,"code")=7209
...
```

If you wanted to log each error message, you could adapt the previous logging example as follows:

```
 If $$$ISERR(tSC) {
   // if error, log error message so users can see them
   Do $SYSTEM.Status.DecomposeStatus(tSC,.errorlist)
   For i=1:1:errorlist {
       Do ..LogMsg(errorlist(i))
   }
 }
```

Note: If you call **DecomposeStatus()** again and pass in the same error array, any new errors are appended to the array.

23.5 Returning a %Status

You can return your own custom status values. To create a %Status, use the following construction:

```
 $$$ERROR($$$GeneralError,"your error text here","parm","anotherparm")
```

Or equivalently:

```
 $SYSTEM.Status.Error($$$GeneralError,"your error text here","parm","anotherparm")
```

Where `"parm"` and `"anotherparm"` represent optional additional error arguments, such as filenames or identifiers for records where the processing did not succeed.

For example:

```
quit $$$ERROR($$$GeneralError,"Not enough information for request")
```

To include information about additional errors, use **$SYSTEM.Status.AppendStatus()** to modify the status value. For example:

```
set tSC=$SYSTEM.Status.AppendStatus(tSCfirst,tSCsecond)
quit tSC
```

23.6 %SYSTEM.Error

The %SYSTEM.Error class is a generic error object. It can be created from a %Status error, from an exception object, a **$ZERROR** error, or an SQLCODE error.

You can use %SYSTEM.Error class methods to convert a %Status to an exception, or to convert an exception to a %Status.

23.7 See Also

For more information, see the class reference for the %SYSTEM.Status class and the %Status (%Library.Status) class.

24
Using TRY-CATCH

Managing the behavior of code when an error (particularly an unexpected error) occurs is called *error handling* or *error processing*. Error handling includes the following operations:

- Correcting the condition that caused the error
- Performing some action that allows execution to resume despite the error
- Diverting the flow of execution
- Logging information about the error

InterSystems IRIS® data platform supports a **TRY-CATCH** mechanism for handling errors. Note that in code migrated from older applications, you might see traditional error processing, which is still fully supported, but is not intended for use in new applications.

Also see %Status Processing, which is not error handling in a strict sense. Typically status processing is fully contained within a **TRY** block.

24.1 Introduction

With **TRY-CATCH**, you can establish delimited blocks of code, each called a **TRY** block; if an error occurs during a **TRY** block, control passes to the **TRY** block's associated **CATCH** block, which contains code for handling the exception. A **TRY** block can also include **THROW** commands; each of these commands explicitly issues an exception from within a **TRY** block and transfers execution to a **CATCH** block.

To use this mechanism in its most basic form, include a **TRY** block within ObjectScript code. If an exception occurs within this block, the code within the associated **CATCH** block is then executed. The form of a **TRY-CATCH** block is:

```
TRY {
     protected statements
} CATCH [ErrorHandle] {
     error statements
}
further statements
```

where:

- The **TRY** command identifies a block of ObjectScript code statements enclosed in curly braces. **TRY** takes no arguments. This block of code is protected code for structured exception handling. If an exception occurs within a **TRY** block, InterSystems IRIS sets the exception properties (oref.Name, oref.Code, oref.Data, and oref.Location), **$ZERROR**, and **$ECODE**, then transfers execution to an exception handler, identified by the **CATCH** command. This is known as throwing an exception.

- The *protected statements* are ObjectScript statements that are part of normal execution. (These can include calls to the **THROW** command. This scenario is described in the following section.)
- The **CATCH** command defines an exception handler, which is a block of code to execute when an exception occurs in a **TRY** block.
- The *ErrorHandle* variable is a handle to an exception object. This can be either an exception object that InterSystems IRIS has generated in response to a runtime error or an exception object explicitly issued by invoking the **THROW** command (described in the next section).
- The *error statements* are ObjectScript statements that are invoked if there is an exception.
- The *further statements* are ObjectScript statements that either follow execution of the *protected statements* if there is no exception or follow execution of the error statements if there is an exception and control passes out of the **CATCH** block.

Depending on events during execution of the protected statements, one of the following events occurs:

- If an error does not occur, execution continues with the *further statements* that appear outside the **CATCH** block.
- If an error does occur, control passes into the **CATCH** block and *error statements* are executed. Execution then depends on contents of the **CATCH** block:
  - If the **CATCH** block contains a **THROW** or **GOTO** command, control goes directly to the specified location.
  - If the **CATCH** block does not contain a **THROW** or **GOTO** command, control passes out of the **CATCH** block and execution continues with the *further statements*.

24.2 Using THROW with TRY-CATCH

InterSystems IRIS issues an implicit exception when a runtime error occurs. To issue an explicit exception, the **THROW** command is available. The **THROW** command transfers execution from the **TRY** block to the **CATCH** exception handler. The **THROW** command has a syntax of:

```
THROW expression
```

where *expression* is an instance of a class that inherits from the %Exception.AbstractException class, which InterSystems IRIS provides for exception handling. For more information on %Exception.AbstractException, see the following section.

The form of the **TRY**/**CATCH** block with a **THROW** is:

```
 TRY {
      protected statements
      THROW expression
      protected statements
 }
 CATCH exception {
      error statements
 }
 further statements
```

where the **THROW** command explicitly issues an exception. The other elements of the **TRY**-**CATCH** block are as described in the previous section.

The effects of **THROW** depends on where the throw occurs and the argument of **THROW**:

- A **THROW** within a **TRY** block passes control to the **CATCH** block.

- A **THROW** within a **CATCH** block passes control up the execution stack to the next error handler. If the exception is a %Exception.SystemException object, the next error handler can be any type (**CATCH** or traditional); otherwise there must be a **CATCH** to handle the exception or a <NOCATCH> error will be thrown.

If control passes into a **CATCH** block because of a **THROW** with an argument, the *ErrorHandle* contains the value from the argument. If control passes into a **CATCH** block because of a system error, the *ErrorHandle* is a %Exception.SystemException object. If no *ErrorHandle* is specified, there is no indication of why control has passed into the **CATCH** block.

For example, suppose there is code to divide two numbers:

```
div(num,div) public {
 TRY {
  SET ans=num/div
 } CATCH errobj {
  IF errobj.Name="<DIVIDE>" { SET ans=0 }
  ELSE { THROW errobj }
 }
 QUIT ans
}
```

If a divide-by-zero error happens, the code is specifically designed to return zero as the result. For any other error, the **THROW** sends the error on up the stack to the next error handler.

24.3 Using $$$ThrowOnError and $$$ThrowStatus Macros

InterSystems IRIS provides macros for use with exception handling. When invoked, these macros throw an exception object to the **CATCH** block.

The following example invokes the `$$$ThrowOnError()` macro when an error status is returned by the **%Prepare()** method:

ObjectScript

```
  #include %occStatus
  TRY {
    SET myquery = "SELECT TOP 5 Name,Hipness,DOB FROM Sample.Person"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET status = tStatement.%Prepare(myquery)
    $$$ThrowOnError(status)
    WRITE "%Prepare succeeded",!
    RETURN
  }
  CATCH sc {
    WRITE "In Catch block",!
    WRITE "error code: ",sc.Code,!
    WRITE "error location: ",sc.Location,!
    WRITE "error data:",$LISTGET(sc.Data,2),!
  RETURN
  }
```

The following example invokes `$$$ThrowStatus` after testing the value of the error status returned by the **%Prepare()** method:

ObjectScript

```
#include %occStatus
TRY {
  SET myquery = "SELECT TOP 5 Name,Hipness,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF ($System.Status.IsError(status)) {
    WRITE "%Prepare failed",!
    $$$ThrowStatus(status) }
  ELSE {WRITE "%Prepare succeeded",!
    RETURN }
}
CATCH sc {
  WRITE "In Catch block",!
  WRITE "error code: ",sc.Code,!
  WRITE "error location: ",sc.Location,!
  WRITE "error data:",$LISTGET(sc.Data,2),!
RETURN
}
```

See System Macros for more information.

24.4 Using the %Exception.SystemException and %Exception.AbstractException Classes

InterSystems IRIS provides the %Exception.SystemException and %Exception.AbstractException classes for use with exception handling. %Exception.SystemException inherits from the %Exception.AbstractException class and is used for system errors. For custom errors, create a class that inherits from %Exception.AbstractException. %Exception.AbstractException contains properties such as the name of the error and the location at which it occurred.

When a system error is caught within a **TRY** block, the system creates a new instance of the %Exception.SystemException class and places error information in that instance. When throwing a custom exception, the application programmer is responsible for populating the object with error information.

An exception object has the following properties:

- Name — The error name, such as <UNDEFINED>
- Code — The error number
- Location — The label+offset^routine location of the error
- Data — Any extra data reported by the error, such as the name of the item causing the error

24.5 Other Considerations with TRY-CATCH

The following describe conditions that may arise when using a **TRY-CATCH** block.

24.5.1 QUIT within a TRY-CATCH Block

A **QUIT** command within a **TRY** or **CATCH** block passes control out of the block to the next statement after the **TRY-CATCH** as a whole.

24.5.2 TRY-CATCH and the Execution Stack

The **TRY** block does not introduce a new level in the execution stack. This means that it is not a scope boundary for **NEW** commands. The error statements execute at the same level as that of the error. This can result in unexpected results if there are **DO** commands within the protected statements and the **DO** target is also within the protected statements. In such cases, the *$ESTACK* special variable can provide information about the relative execution levels.

24.5.3 Using TRY-CATCH with Traditional Error Processing

TRY-CATCH error processing is compatible with **$ZTRAP** error traps used at different levels in the execution stack. The exception is that **$ZTRAP** may not be used within the protected statements of a **TRY** clause. User-defined errors with a **THROW** are limited to **TRY-CATCH** only. User-defined errors with the **ZTRAP** command may be used with any type of error processing.

25 Error Logging

Each namespace can have an application error log, which records errors encountered when running code in that namespace. Some system code automatically writes to this log, and your code can do so as well.

25.1 Logging Application Errors

To log an exception to the application error log, use the **%Exception.AbstractException.Log()** method. Typically you would do this within the CATCH block of a TRY-CATCH.

25.2 Using Management Portal to View Application Error Logs

From the Management Portal, select **System Operation**, then **System Logs**, then **Application Error Log**. This displays the **Namespace** list of those namespaces that have application error logs. You can use the header to sort the list.

Select **Dates** for a namespace to display those dates for which there are application error logs, and the number of errors recorded for that date. You can use the headers to sort the list. You can use **Filter** to match a string to the Date and Quantity values.

Select **Errors** for a date to display the errors for that date. Error # integers are assigned to errors in chronological order. Error # `*COM` is a user comment applied to all errors for that date. You can use the headers to sort the list. You can use **Filter** to match a string.

Select **Details** for an error to open an **Error Details** window that displays state information at the time of the error including special variables values and **Stacks** details. To see the stack trace corresponding to the error, click the **Stacks** or scroll to the bottom of the page. Then click the + icon in the row of this table and look for the *%objlasterror* variable, which (if present) contains information about the error.

You can specify a user comment for an individual error.

The **Namespaces**, **Dates**, and **Errors** listings include check boxes that allow you to delete the error log for the corresponding error or errors. Check what you wish to delete, then select the **Delete** button.

25.3 Using ^%ERN to View Application Error Logs

The **^%ERN** utility examines application errors and lets you see all errors logged for the current namespace. This is an alternative to using the Management Portal.

Take the following steps to use the **^%ERN** utility:

1. In an ObjectScript shell, enter **DO ^%ERN**. The name of the utility is case-sensitive; responses to prompts within the utility are not case-sensitive.

   At any prompt you may enter `?` to list syntax options for the prompt, or `?L` to list all of the defined values. You may use the **Enter** key to exit to the previous level.

2. At the `For Date:` prompt, enter `?L` to see a list of all the dates when errors occurred.

3. Then at the same prompt, enter one of those dates (in the format mm/dd/yyyy); if you omit the year, the current year is assumed. The routine then displays the date and the number of errors logged for that date. Alternative, you can retrieve lists of errors from this prompt using the following syntax:

   - `?L` lists all dates on which errors occurred, most recent first, with the number of errors logged. The `(T)` column indicates how many days ago, with `(T)` = today and `(T-7)` = seven days ago. If a user comment is defined for all of the day's errors, it is shown in square brackets. After listing, it re-displays the `For Date:` prompt. You can enter a date or `T-`*`n`*.
   - `[text` lists all errors that contain the substring *text*. `<text` lists all errors that contain the substring *text* in the error name component. `^text` lists all errors that contain the substring *text* in the error location component. After listing, it re-displays the `For Date:` prompt. Enter a date.

4. `Error:` at this prompt supply the integer number for the error you want to examine: 1 for the first error of the day, 2 for the second, and so on. Or enter a question mark (?) for a list of available responses. The utility displays the following information about the error: the Error Name, Error Location, time, system variable values, and the line of code executed at the time of the error.

   You can specify an `*` at the `Error:` prompt for comments. `*` displays the current user-specified comment applied to all of the errors of that day. It then prompts you to supply a new comment to replace the existing comment for all of these errors.

5. `Variable:` at this prompt you can specify numerous options for information about variables. If you specify the name of a local variable (unsubscipted or subscripted), **^%ERN** returns the stack level and value of that variable (if defined), and all its descendent nodes. You cannot specify a global variable, process-private variable, or special system variable.

   You may enter `?` to list other syntax options for the `Variable:` prompt.

   - `*A`: when specified at the `Variable:` prompt, displays the `Device:` prompt; press **Return** to display results.
   - `*V`: when specified at the `Variable:` prompt, displays the `Variable(s):` prompt. At this prompt specify an unsubscripted local variable or a comma-separated list of unsubscripted local variables; subscripted variables are rejected. **^%ERN** then displays the `Device:` prompt; press **Return** to display results. **^%ERN** returns the value of each specified variable (if defined) and all its descendent nodes.
   - `*L`: when specified at the `Variable:` prompt, loads the variables into the current partition. It loads all private variables (as public) and then all public variables that don't conflict with the loaded private variables.

25.4 See Also

- **%SYS.ProcessQuery.ExamStackByPid()** method, which provides details on the ^*mtemp* global used by **^%ERN**

26 Command-Line Routine Debugging

This topic describes techniques for testing and debugging Object Script code. InterSystems IRIS® data platform gives you two ways to debug code:

- Use the BREAK command in routine code to suspend execution and allow you to examine what is happening.
- Use the ZBREAK command to invoke the ObjectScript Debugger to interrupt execution and allow you to examine both code and variables.

InterSystems IRIS includes the ability to suspend a routine and enter a shell that supports full debugging capabilities, as described in this topic. InterSystems IRIS also includes a secure debug shell, which has the advantage of ensuring that users are prevented from exceeding or circumventing their assigned privileges.

26.1 Secure Debug Shell

The secure debug shell helps better control access to sensitive data. It is an environment that allows users to perform basic debugging, such as stepping and displaying variables, but does not allow them to do anything that changes the execution path or results of a routine. This protects against access that can lead to issues such as manipulation, malicious role escalation, and the injection of code to run with higher privileges.

By default, users at the debug prompt maintain their current level of privileges. To enable the secure shell for the debug prompt and thereby restrict the commands that the user may issue, you must enable the secure debug shell for that user.

If enabled for the current user, the secure debug shell starts when a **BREAK** command is executed, a breakpoint or watchpoint is encountered, or an uncaught error is issued.

Within the secure debug shell, the user cannot invoke:

- Any command that can modify a variable.
- Any function that can modify a variable.
- Any command that can call other routines.
- Any command that affects the flow of the routine or the environment.

Within the secure debug shell, when a user attempts to invoke a restricted command or function, InterSystems IRIS throws a <COMMAND> or <FUNCTION> error, respectively.

26.1.1 Restricted Commands and Functions

This section lists the restricted activities within the secure debug shell:

- Restricted ObjectScript Commands
- Restricted ObjectScript Functions
- Restricted Object Constructions

26.1.1.1 Restricted ObjectScript Commands

The following are the restricted ObjectScript commands for the secure debug shell:

- **CLOSE**
- **DO**
- **FOR**
- **GOTO** with an argument
- **KILL**
- **LOCK**
- **MERGE**
- **OPEN**
- **QUIT**
- **READ**
- **RETURN**
- **SET**
- **TCOMMIT**
- **TROLLBACK**
- **TSTART**
- **VIEW**
- **XECUTE**
- **ZINSERT**
- **ZKILL**
- **ZREMOVE**
- **ZSAVE**
- user commands except **ZW** and **ZZDUMP**

26.1.1.2 Restricted ObjectScript Functions

The following are the restricted ObjectScript functions for the secure debug shell:

- **$CLASSMETHOD**
- **$COMPILE**

- **$DATA(,var)** — two-argument version only
- **$INCREMENT**
- **$METHOD**
- **$ORDER(,,var)** — three-argument version only
- **$PROPERTY**
- **$QUERY(,,var)** — three-argument version only
- **$XECUTE**
- **$ZF**
- **$ZSEEK**
- any extrinsic function

26.1.1.3 Restricted Object Constructions

No method or property references are allowed. Property references are restricted because they could invoke a **propertyGet** method. Some examples of the object method and property syntax constructions that are restricted are:

- **#class(classname).ClassMethod()**
- **oref.Method()**
- oref.Property
- **$SYSTEM.Class.Method()**
- **..Method()**
- ..Property

Note: Even without passing a variable by reference, a method can modify public variables. Since a property reference could invoke a **propGet** method, no property access is allowed.

26.2 Debugging with the ObjectScript Debugger

The ObjectScript Debugger lets you test routines by inserting debugging commands directly into your routine code. Then, when you run the code, you can issue commands to test the conditions and the flow of processing within your application. Its major capabilities are:

- Set breakpoints with the **ZBREAK** command at code locations and take specified actions when those points are reached.
- Set watchpoints on local variables and take specified actions when the values of those variables change.
- Interact with InterSystems IRIS during a breakpoint/watchpoint in a separate window.
- Trace execution and output a trace record (to a terminal or other device) whenever the path of execution changes.
- Display the execution stack.
- Run an application on one device while debugging I/O goes to a second device. This enables full screen InterSystems IRIS applications to be debugged without disturbing the application’s terminal I/O.

26.2.1 Using Breakpoints and Watchpoints

The ObjectScript Debugger provides two ways to interrupt program execution:

- *Breakpoints*
- *Watchpoints*

A breakpoint is a location in an InterSystems IRIS routine that you specify with the **ZBREAK** command. When routine execution reaches that line, InterSystems IRIS suspends execution of the routine and, optionally, executes debugging actions you define. You can set breakpoints in up to 20 routines. You can set a maximum of 20 breakpoints within a particular routine.

A watchpoint is a variable you identify in a **ZBREAK** command. When its value is changed with a **SET** or **KILL** command, you can cause the interruption of routine execution and/or the execution of debugging actions you define within the **ZBREAK** command. Note that you cannot set watchpoints for system variables.

Breakpoints and watchpoints you define are not maintained from one session to another. Therefore, you may find it useful to store breakpoint/watchpoint definitions in a routine or **XECUTE** command string so it is easy to reinstate them between sessions.

26.2.2 Establishing Breakpoints and Watchpoints

You use the ZBREAK command to establish breakpoints and watchpoints.

26.2.2.1 Syntax

```
ZBREAK location[:action:condition:execute_code]
```

where:

Argument	Description
location	Required. Specifies a code location (that sets a breakpoint) or local or system variable (which sets a watchpoint). If the location specified already has a breakpoint/watchpoint defined, the new specification completely replaces the old one. Note that you cannot watchpoints for system variables.
action	*Optional* — Specifies the action to take when the breakpoint/watchpoint is triggered. For breakpoints, the action occurs before the line of code is executed. For watchpoints, the action occurs after the command that modifies the local variable. Actions may be upper- or lowercase, but must be enclosed in quotation marks.
condition	*Optional* — A boolean expression, enclosed in curly braces or quotes, that is evaluated when the breakpoint/watchpoint is triggered. • When *condition* is true (1), the action is carried out. • When *condition* is false, the *action* is not carried out and the code in *execute_code* is not executed. If *condition* is not specified, the default is true.
execute_code	*Optional* — Specifies ObjectScript code to be executed if *condition* is true. If the code is a literal, it must be surrounded by curly braces or quotation marks. This code is executed before the action being carried out. Before the code is executed, the value of the $TEST special system variable is saved. After the code has executed, the value of **$TEST** as it existed in the program being debugged is restored.

Note: Using **ZBREAK** with a ? (question mark) displays help.

26.2.2.2 Setting Breakpoints with Code Locations

You specify code locations as a routine line reference that you can use in a call to the $TEXT function. A breakpoint occurs whenever execution reaches this point in the code, before the execution of the line of code. If you do not specify a routine name, InterSystems IRIS assumes the reference is to the current routine.

26.2.2.3 Argumentless GOTO in Breakpoint Execution Code

An argumentless GOTO is allowed in breakpoint execution code. Its effect is equivalent to executing an argumentless **GOTO** at the debugger **BREAK** prompt and execution proceeds until the next breakpoint.

For example, if the routine you are testing is in the current namespace, you can enter location values such as these:

Value	Break Location
label^*rou*	Break before the line at the line label *label* in the routine *rou*.
label+3^*rou*	Break before the third line after the line label *label* in routine *rou*.
+3^*rou*	Break before the third line in routine *rou*.

If the routine you are testing is currently loaded in memory (that is, an implicit or explicit ZLOAD was performed), you can use location values such as these:

Value	Break Location
label	Break before the line label at *label*.
label+3	Break before the third line after *label*.
+3	Break before the third line.

26.2.2.4 Setting Watchpoints with Local and System Variable Names

Local variable names cause a watchpoint to occur in these situations:

- When the local variable is created
- When a **SET** command changes the value of the local variable
- When a **KILL** command deletes the local variable

Variable names are preceded by an asterisk, as in `*a`.

If you specify an array-variable name, the ObjectScript Debugger watches all descendant nodes. For instance, if you establish a watchpoint for array *a*, a change to a(5) or a(5,1) triggers the watchpoint.

The variable need not exist when you establish the watchpoint.

You can also use the following special system variables:

System Variable	Trigger Event
$ZERROR	Triggered whenever an error occurs, before invoking the error trap.
$ZTRAP	Triggered whenever an error trap is set or cleared.
$IO	Triggered whenever explicitly SET.

26.2.2.5 Action Argument Values

The following table describes the values you can use for the **ZBREAK** *action* argument.

Argument	Description
"B"	Default, except if you include the "T" action, then you must also explicitly include the "B" action, as in ZBREAK *a:"TB", to actually cause a break. Suspends execution and displays the line at which the break occurred along with a caret (^) indicating the point in the line. Then displays the Terminal prompt and allows interaction. Execution resumes with an argumentless **GOTO** command.
"L"	Same as "B", except **GOTO** initiates single-step execution, stopping at the beginning of each line. When a **DO** command, user-defined function, or **XECUTE** command is encountered, single-step mode is suspended until that command or function completes.
"L+"	Same as "B", except **GOTO** initiates single-step execution, stopping at the beginning of each line. **DO** commands, user-defined functions, and **XECUTE** commands do not suspend single-step mode.
"S"	Same as "B", except **GOTO** initiates single-step execution, stopping at the beginning of each command. When a **DO** command, user-defined function, **FOR** command, or **XECUTE** command is encountered, single-step mode is suspended until that command or function completes.

Argument	Description
"S+"	Same as "B", except **GOTO** initiates single-step execution, stopping at the beginning of each command. **DO** commands, user-defined functions, **FOR** commands, and **XECUTE** commands do not suspend single-step mode.
"T"	Can be used together with any other argument. Outputs a trace message to the trace device. This argument works only after you have set tracing to be ON with the ZBREAK /TRACE:ON command, described later. The trace device is the principal device unless you define it differently in the ZBREAK /TRACE command. If you use this argument with a breakpoint, you see the following message: TRACE: ZBREAK at label2^rou2. If you use this argument with a watchpoint, you see a trace message that names the variable being watched and the command being acted upon. In the example below, the variable *a* was being watched. It changed at the line test+1 in the routine test. TRACE: ZBREAK SET a=2 at test+1^test. If you include the "T" action, you must also explicitly include the "B" action as in ZBREAK *a:"TB", to have an actual break occur.
"N"	Take no action at this breakpoint/watchpoint. The *condition* expression is always evaluated and determines if the *execute_code* is executed.

26.2.2.6 ZBREAK Examples

The following example establishes a watchpoint that suspends execution whenever the local variable *a* is killed. No action is specified, so "B" is assumed.

```
ZBREAK *a::"'$DATA(a)"
```

The following example illustrates the above watchpoint acting on a direct mode ObjectScript command (rather than on a command issued from within a routine). The caret (^) points to the command that caused execution to be suspended:

Terminal

```
USER>KILL a
KILL a
^
<BREAK>
USER 1s0>
```

The following example establishes a breakpoint that suspends execution and sets single-step mode at the beginning of the line *label2^rou*.

```
ZBREAK label2^rou:"L"
```

The following example shows how the break would appear when the routine is run. The caret (^) indicates where execution was suspended.

Terminal

```
USER>DO ^rou
label2 SET x=1
 ^
<BREAK>label2^rou
USER 2d0>
```

In the following example, a breakpoint at line *label3^rou* does not suspend execution, because of the "N" action. However, if x<1 when the line *label3^rou* is reached, then *flag* is SET to *x*.

```
ZBREAK label3^rou:"N":"x<1":"SET flag=x"
```

The following example establishes a watchpoint that executes the code in ^GLO whenever the value of *a* changes. The double colon indicates no *condition* argument.

```
ZBREAK *a:"N"::"XECUTE ^GLO"
```

The following example establishes a watchpoint that causes a trace message to display whenever the value of *b* changes. The trace message will display only if trace mode has been turned on with the **ZBREAK /TRACE:ON** command.

```
ZBREAK *b:"T"
```

The following example establishes a watchpoint that suspends execution in single-step mode when variable *a* is set to 5.

```
ZBREAK *a:"S":"a=5"
```

When the break occurs in the following example, a caret (^) symbol points to the command that caused the variable *a* to be set to 5.

Terminal

```
USER>DO ^test
FOR i=1:1:6 SET a=a+1
  ^
<BREAK>
test+3^test
USER 3f0>WRITE a
5
```

26.2.3 Disabling Breakpoints and Watchpoints

You can disable either:

- Specific breakpoints and watchpoints
- All breakpoints or watchpoints

26.2.3.1 Disabling Specific Breakpoints and Watchpoints

You can disable a breakpoint or watchpoint by preceding the location with a minus sign. The following command disables a breakpoint previously specified for location *label2^rou*:

```
ZBREAK -label2^rou
```

A disabled breakpoint is “turned off”, but InterSystems IRIS remembers its definition. You can enable the disabled breakpoint by preceding the location with a plus sign. The following command enables the previously disabled breakpoint:

```
ZBREAK +label2^rou
```

26.2.3.2 Disabling All Breakpoints and Watchpoints

You can disable all breakpoints or watchpoints by using the plus or minus signs without a location:

Sign	Description
ZBREAK -	Disable all defined breakpoints and watchpoints.
ZBREAK +	Enable all defined breakpoints and watchpoint.

26.2.4 Delaying Execution of Breakpoints and Watchpoints

You can also delay the execution of a break/watchpoint for a specified number of iterations. You might have a line of code that appears within a loop that you want to break on periodically, rather than every time it is executed. To do so, establish the breakpoint as you would normally, then disable with a count following the location argument.

The following **ZBREAK** command causes the breakpoint at *label2^rou* to be disabled for 100 iterations. On the 101st time this line is executed, the specified breakpoint action occurs.

```
 ZBREAK label2^rou       ; establish the breakpoint
 ZBREAK -label2^rou#100  ; disable it for 100 iterations
```

Important: A delayed breakpoint is not decremented when a line is repeatedly executed because it contains a **FOR** command.

26.2.5 Deleting Breakpoints and Watchpoints

You can delete individual break/watchpoints by preceding the location with a double minus sign; for example:

```
ZBREAK --label2^rou
```

After you have deleted a breakpoint/watchpoint, you can only reset it by defining it again.

To delete all breakpoints, issue the command:

```
ZBREAK /CLEAR
```

This command is performed automatically when an InterSystems IRIS process halts.

26.2.6 Single-step Breakpoint Actions

You can use single step execution to stop execution at the beginning of each line or of each command in your code. You can establish a single step breakpoint to specify actions and execution code to be executed at each step. Use the following syntax to define a single step breakpoint:

```
ZBREAK $:action[:condition:execute_code]
```

Unlike other breakpoints, **ZBREAK** $ does not cause a break, because breaks occur automatically as you single-step. **ZBREAK** $ lets you specify actions and execute code at each point where the debugger breaks as you step through the routine. It is especially useful in tracing executed lines or commands. For example, to trace executed lines in the application ^TEST:

Terminal

```
USER>ZBREAK /TRACE:ON
USER>BREAK "L+"
USER>ZBREAK $:"T"
```

The "T" action specified alone (that is, without any other action code) suppresses the single step break that normally occurs automatically. (You can also suppress the single-step break by specifying the "N" action code — either with or without any other action codes.)

Establish the following single-step breakpoint definition if both tracing and breaking should occur:

Terminal

```
USER>ZBREAK $:"TB"
```

26.2.7 Tracing Execution

You can control whether or not the "T" action of the **ZBREAK** command is enabled by using the following form of **ZBREAK**:

ZBREAK /TRACE:*state*[:*device*]

where *state* can be:

State	Description
ON	Enables tracing.
OFF	Disables tracing.
ALL	Enables tracing of application by performing the equivalent of: ZBREAK /TRACE:ON[:device] BREAK "L+" ZBREAK $:"T"

When *device* is used with the ALL or ON state keywords, trace messages are redirected to the specified device rather than to the principal device. If the device is not already open, InterSystems IRIS attempts to open it as a sequential file with WRITE and APPEND options.

When device is specified with the OFF state keyword, InterSystems IRIS closes the file if it is currently open.

Note: `ZBREAK /TRACE:OFF` does not delete or disable the single-step breakpoint definition set up by `ZBREAK /TRACE:ALL`, nor does it clear the `L+` single stepping set up by `ZBREAK /TRACE:ALL`. You must also issue the commands `ZBREAK --$` and `BREAK "C"` to remove the single stepping; alternatively, you can use the single command `BREAK "OFF"` to turn off all debugging for the process.

Tracing messages are generated at breakpoints associated with a `T` action. With one exception, the trace message format is as follows for all breakpoints:

Trace: ZBREAK at *line_reference*

where *line_reference* is the line reference of the breakpoint.

The trace message format is slightly different for single step breakpoints when stepping is done by command:

Trace: ZBREAK at *line_reference source_offset*

where *line_reference* is the line reference of the breakpoint and *source_offset* is the 0-based offset to the location in the source line where the break has occurred.

Operating System Notes:

- *Windows* — Trace messages to another device are supported on Windows platforms for terminal devices connected to a COM port, such as COM1:. You cannot use the console or a terminal window. You can specify a sequential file for the trace device
- *UNIX®* — To send trace messages to another device on UNIX® platforms:
  1. Log in to /dev/tty01.
  2. Verify the device name by entering the tty command:

     ```
     $ tty
     /dev/tty01
     ```
  3. Issue the following command to avoid contention for the device:

     ```
     $ exec sleep 50000
     ```

4. Return to your working window.
5. Start and enter InterSystems IRIS.
6. Issue your trace command:

   ```
   ZBREAK /T:ON:"/dev/tty01"
   ```

7. Run your program.

   If you have set breakpoints or watchpoints with the T action, you see trace messages appear in the window connected to /dev/tty01.

26.2.7.1 Trace Message Format

If you set a code breakpoint, the following message appears:

```
Trace: ZBREAK at label2^rou2
```

If you set a variable watchpoint, one of the following messages appears:

```
Trace: ZBREAK SET var=val at label2^rou2
Trace: ZBREAK SET var=Array Val at label2^rou2
Trace: ZBREAK KILL var at label2^rou2
```

- *var* is the variable being watched.
- *val* is the new value being set for that variable.

If you issue a **NEW** command, you receive no trace message. However, the trace on the variable is triggered the next time you issue a **SET** or **KILL** on the variable at the **NEW** level. If a variable is passed by reference to a routine, then that variable is still traced, even though the name has effectively changed.

26.2.8 INTERRUPT Keypress and Break

Normally, pressing the interrupt key sequence (typically **CTRL-C**) generates a trapable (<INTERRUPT>) error. To set interrupt processing to cause a break instead of an <INTERRUPT> error, use the following **ZBREAK** command: **`ZBREAK /INTERRUPT:Break`**

This causes a break to occur when you press the INTERRUPT key even if you have disabled interrupts at the application level for the device.

If you press the INTERRUPT key during a read from the terminal, you may have to press **RETURN** to display the break-mode prompt. To reset interrupt processing to generate an error rather than cause a break, issue the following command: **`ZBREAK /INTERRUPT:NORMAL`**

26.2.9 Displaying Information About the Current Debug Environment

To display information about the current debug environment, including all currently defined break or watchpoints, issue the **ZBREAK** command with no arguments.

The argumentless **ZBREAK** command describes the following aspects of the debug environment:

- Whether **CTRL-C** causes a break
- Whether trace output specified with the "T" action in the **ZBREAK** command displays
- The location of all defined breakpoints, with flags describing their enabled/disabled status, action, condition and executable code

- All variables for which there are watchpoints, with flags describing their enabled/disabled status, action, condition and executable code

Output from this command is displayed on the device you have defined as your debug device, which is your principal device unless you have defined the debug device differently with the ZBREAK /DEBUG command described in the Using the Debug Device section.

The following table describes the flags provided for each breakpoint and watchpoint:

Display Section	Meaning
Identification of break/watchpoint	Line in routine for breakpoint. Local variable for watchpoint.
F:	Flag providing information about the type of action defined in the **ZBREAK** command.
S:	Number of iterations to delay execution of a breakpoint/watchpoint defined in a **ZBREAK -** command.
C:	Condition argument set in **ZBREAK** command.
E:	Execute_code argument set in **ZBREAK** command.

The following table describes how to interpret the F: value in a breakpoint/watchpoint display. The F: value is a list of the applicable values in the first column.

Value	Meaning
E	Breakpoint or watchpoint enabled
D	Breakpoint or watchpoint disabled
B	Perform a break
L	Perform an "L"
L+	Perform an "L+"
S	Perform an "S"
S+	Perform an "S+"
T	Output a Trace Message

26.2.9.1 Default Display

When you first enter InterSystems IRIS and use **ZB**, the output is as follows:

Terminal

```
USER>ZBREAK
BREAK:
No breakpoints
No watchpoints
```

This means:

- Trace execution is OFF
- There is no break if **CTRL-C** is pressed

- No break/watchpoints are defined

26.2.9.2 Display When Breakpoints and Watchpoints Exist

This example shows two breakpoints and one watchpoint being defined:

Terminal

```
USER>ZBREAK +3^test:::{WRITE "IN test"}
USER>ZBREAK -+3^test#5
USER>ZBREAK +5^test:"L"
USER>ZBREAK -+5^test
USER>ZBREAK *a:"T":"a=5"
USER>ZBREAK /TRACE:ON
USER>ZBREAK
BREAK: TRACE ON
+3^test F:EB S:5 C: E:"WRITE ""IN test"""
+5^test F:DL S:0 C: E:
a F:ET S:0 C:"a=5" E:
```

The first two **ZBREAK** commands define a delayed breakpoint; the second two **ZBREAK** commands define a disabled breakpoint; the fifth **ZBREAK** command defines a watchpoint. The sixth **ZBREAK** command enables trace execution. The final **ZBREAK** command, with no arguments, displays information about current debug settings.

In the example, the **ZBREAK** display shows that:

- Tracing is ON
- There is no break if **CTRL-C** is pressed.

The output then describes the two breakpoints and one watchpoint:

- The F flag for the first breakpoint equals `EB` and the S flag equals 5, which means that a breakpoint will occur the fifth time the line is encountered. The E flag displays executable code, which will run before the Terminal prompt for the break is displayed.
- The F flag for the second breakpoint equals `DL`, which means it is disabled, but if enabled will break and then single-step through each line of code following the breakpoint location.
- The F flag for the watchpoint is `ET`, which means the watchpoint is enabled. Since trace execution is ON, trace messages will appear on the trace device. Because no trace device was defined, the trace device will be the principal device.
- The C flag means that trace is displayed only when *condition* is true.

26.2.10 Using the Debug Device

The debug device is the device where:

- The **ZBREAK** command displays information about the debug environment.
- The Terminal prompt appears if a break occurs.

Note: On Windows platforms, trace messages to another device are supported only for terminal devices connected to a COM port, such as COM1:

When you enter InterSystems IRIS, the debug device will automatically be set to your principal device. At any time, debugging I/O can be sent to an alternate device with the command: `ZBREAK /DEBUG:"`*`device`*`"`.

Note: There are also operating-system-specific actions that you can take.

On UNIX® systems, to cause the break to occur on the tty01 device, issue the following command:

```
ZBREAK /D:"/dev/tty01/"
```

When a break occurs, because of a **CTRL-C** or to a breakpoint or watchpoint being triggered, it appears in the window connected to the device. That window becomes the active window.

If the device is not already open, an automatic OPEN is performed. If the device is already open, any existing OPEN parameters are respected.

Important: If the device you specify is not an interactive device (such as a terminal), you may not be able to return from a break. However, the system does not enforce this restriction.

26.2.11 ObjectScript Debugger Example

First, suppose you are debugging the simple program named test shown below. The goal is to put 1 in variable *a*, 2 in variable *b*, and 3 in variable *c*.

```
test; Assign the values 1, 2, and 3 to the variables a, b, and c
 SET a=1
 SET b=2
 SET c=3 KILL a WRITE "in test, at end"
 QUIT
```

However, when you run test, only variables *b* and *c* hold the correct values:

Terminal

```
USER>DO ^test
in test, at end
USER>WRITE
b=2
c=3
USER>
```

The problem in the program is obvious: variable *a* is KILLed on line 4. However, assume you need to use the debugger to determine this.

You can use the **ZBREAK** command to set single-stepping through each line of code ("L" action) in the routine test. By a combination of stepping and writing the value of *a*, you determine that the problem lies in line 4:

Terminal

```
USER>NEW
USER 1S1>ZBREAK
BREAK
No breakpoints
No watchpoints
USER 1S1>ZBREAK ^test:"L"
USER 1S1>DO ^test
SET a=1
^
<BREAK>test+1^test
USER 3d3>WRITE a
<UNDEFINED>^test
USER 3d3>GOTO
SET b=2
^
<BREAK>test+2^test
USER 3d3>WRITE a
1
USER 3d3>GOTO
SET c=3 KILL a WRITE "in test, at end"
^
<BREAK>test+3^test
USER 3d3>WRITE a
```

```
1
USER 3d3>GOTO
in test, at end
QUIT
^
<BREAK>test+4^test
USER 3d3>WRITE a
WRITE a
^
<UNDEFINED>^test
USER 3d3>GOTO
USER 1S1>
```

You can now examine that line and notice the **KILL a** command. In more complex code, you might now want to single-step by command ("S" action) through that line.

If the problem occurred within a **DO**, **FOR**, or **XECUTE** command or a user-defined function, you would use the "L+" or "S+" actions to single-step through lines or commands within the lower level of code.

26.2.12 Understanding ObjectScript Debugger Errors

The ObjectScript Debugger flags an error in a condition or execute argument with an appropriate InterSystems IRIS error message.

If the error is in the *execute_code* argument, the condition surrounds the execute code when the execute code is displayed before the error message. The condition special variable ($TEST) is always set back to 1 at the end of the execution code so that the rest of the debugger processing code works properly. When control returns to the routine, the value of **$TEST** within the routine is restored.

Suppose you issue the following **ZBREAK** command for the example program test:

Terminal

```
USER>ZBREAK test+1^test:"B":"a=5":"WRITE b"
```

In the program test, variable *b* is not defined at line test+1, so there is an error. The error display appears as follows:

```
IF a=5 XECUTE "WRITE b" IF 1
^
<UNDEFINED>test+1^test
```

If you had not defined a *condition*, then an artificial true condition would be defined before and after the execution code; for example:

Terminal

```
USER>IF 1 WRITE b IF 1
```

26.3 Debugging With BREAK

InterSystems IRIS includes three forms of the BREAK command:

- **BREAK** without an argument inserted into routine code establishes a breakpoint at that location. When encountered during code execution this breakpoint suspend execution and returns to the Terminal prompt.
- **BREAK** with a letter string argument establishes or deletes breakpoints at that enable stepping through code on a line-by-line or command-by-command basis.
- The **BREAK** command with an integer argument enables or disables **CTRL-C** user interrupts. (Refer to the BREAK command for further details.)

26.3.1 Using Argumentless BREAK to Suspend Routine Execution

To suspend a running routine and return the process to the Terminal prompt, enter an argumentless BREAK into your routine at points where you want execution to temporarily stop.

When InterSystems IRIS encounters a **BREAK**, it takes the following steps:

1. Suspends the running routine
2. Returns the process to the Terminal prompt. When debugging an application that uses I/O redirection of the principal device, redirection will be turned off at the debug prompt so output from a debug command will be shown on the Terminal.

   You can now issue ObjectScript commands, modify data, and execute further routines or subroutines, even those with errors or additional BREAKs. If you issue an ObjectScript command from the debug Terminal prompt, this command is immediately executed. It is not inserted into the running routine. This command execution is the same behavior as the ordinary Terminal prompt, with one difference: a command proceeded by a Tab character is executed from the debug Terminal prompt; a command proceeded by a Tab character is not executed from the ordinary Terminal prompt.

To resume execution at the point at which the routine was suspended, issue an argumentless **GOTO** command.

You may find it useful to specify a postconditional on an argumentless BREAK command so that you can rerun the same code simply by setting the postconditional variable rather than having to change the routine. For example, you may have the following line in a routine:

```
CHECK BREAK:$DATA(debug)
```

You can then set the variable *debug* to suspend the routine and return the job to the Terminal prompt or clear the variable *debug* to continue running the routine.

For further details, see Command Postconditional Expressions.

26.3.2 Using Argumented BREAK to Suspend Routine Execution

You do not have to place argumentless **BREAK** commands at every location where you want to suspend your routine. InterSystems IRIS provides several argument options that allow you to step through the execution of the code. You can step through the code by single steps (**BREAK “S”**) or by command line (**BREAK “L”**). For a full list of these letter code arguments, see the BREAK command.

One difference between **BREAK “S”** and **BREAK “L”** is that many command lines consist of more than one step. This is not always obvious. For example, the following are all one line (and one ObjectScript command), but each is parsed as two steps: **SET x=1,y=2**, **KILL x,y**, **WRITE “hello”,!**, **IF x=1,y=2**.

Both **BREAK “S”** and **BREAK “L”** ignore label lines, comments, and **TRY** statements (though both break at the closing curly brace of a **TRY** block). **BREAK “S”** breaks at a **CATCH** statement (if the **CATCH** block is entered); **BREAK “L”** does not.

When a **BREAK** returns the process to the Terminal prompt, the break state is not stacked. Thus you can change the break state and the new state remains in effect when you issue an argumentless **GOTO** to return to the executing routine.

InterSystems IRIS stacks the break state whenever a **DO**, **XECUTE**, **FOR**, or user-defined function is entered. If you choose **BREAK "C"** to turn off breaking, the system restores the break state at the end of the **DO**, **XECUTE**, **FOR**, or user-defined function. Otherwise, InterSystems IRIS ignores the stacked state.

Thus if you enable breaking at a low subroutine level, breaking continues after the routine returns to a higher subroutine level. In contrast, if you disable breaking at a low subroutine level that was in effect at a higher level, breaking resumes when you return to that higher level. You can use **BREAK "C-"** to disable breaking at all levels.

You can use **BREAK "L+"** or **BREAK "S+"** to enable breaking within a **DO**, **XECUTE**, **FOR**, or a user-defined function.

You can use **BREAK "L-"** to disable breaking at the current level but enables line breaking at the previous level. You can use **BREAK "S-"** to disable breaking at the current level but enables single-step breaking at the previous level.

26.3.2.1 Shutting Off Debugging

To remove all debugging that has been established for a process, use the `BREAK "OFF"` command. This command removes all breakpoints and watchpoints and turns off stepping at all program stack levels. It also removes the association with the debug and trace devices, but does not close them.

Invoking `BREAK "OFF"` is equivalent to issuing the following set of commands:

ObjectScript

```
ZBREAK /CLEAR
ZBREAK /TRACE:OFF
ZBREAK /DEBUG:""
ZBREAK /ERRORTRAP:ON
BREAK "C-"
```

26.3.3 Terminal Prompt Shows Program Stack Information

When a BREAK command suspends execution of a routine or when an error occurs, the program stack retains some stacked information. When this occurs, a brief summary of this information is displayed as part of the Terminal prompt (*namespace*>). For example, this information might take the form: `USER 5d3>`, where:

Character	Description
5	Indicates there are five stack levels. A stack level can be caused by a **DO**, **FOR**, **XECUTE**, **NEW**, user-defined function call, error state, or break state.
d	Indicates that the last item stacked is a **DO**.
3	Indicates there are 3 **NEW** states, parameter passing, or user-defined functions on the stack. This value is a zero if no **NEW** commands, parameter passing, or user-defined functions are stacked.

Terminal prompt letter codes are listed in the following table.

Table 26–1: Stack Error Codes at the Terminal Prompt

Prompt	Definition
d	**DO**
e	user-defined function
f	**FOR** loop
x	**XECUTE**
B	**BREAK** state
E	Error state
N	**NEW** state
S	Sign-on state

In the following example, command line statements are shown with their resulting Terminal prompts when adding stack frames:

Terminal

```
USER>NEW
USER 1S1>NEW
USER 2N1>XECUTE "NEW  WRITE 123 BREAK"
<BREAK>
USER 4x1>NEW
USER 5B1>BREAK
<BREAK>
USER 6N2>
```

You can unwind the program stack using **QUIT 1**. The following is an example of Terminal prompts when unwinding the stack:

Terminal

```
USER 6f0>QUIT 1  /* an error occurred in a FOR loop. */
USER 5x0>QUIT 1  /* the FOR loop was in code invoked by XECUTE. */
USER 4f0>QUIT 1  /* the XECUTE was in a FOR loop. */
USER 3f0>QUIT 1  /* that FOR loop was nested inside another FOR loop. */
USER 2d0>QUIT 1  /* the DO command was used to execute the program. */
USER 1S0>QUIT 1  /* sign on state. */
USER>
```

26.3.4 FOR Loop and WHILE Loop

You can use either a FOR or a WHILE to perform the same operation: loop until an event (usually a counter increment) causes execution to break out of the loop. However, which loop construct you use has consequences for performing single-step (**BREAK "S+"** or **BREAK "L+"**) debugging on the code module.

A **FOR** loop pushes a new level onto the stack. A **WHILE** loop does not change the stack level. When debugging a **FOR** loop, popping the stack from within the **FOR** loop (using `BREAK "C" GOTO` or `QUIT 1`) allows you to continue single-step debugging with the command immediately following the end of the **FOR** command construct. When debugging a **WHILE** loop, issuing a using `BREAK "C" GOTO` or `QUIT 1` does not pop the stack, and therefore single-step debugging does not continue following the end of the **WHILE** command. The remaining code executes without breaking.

26.3.5 Resuming Execution after a BREAK or an Error

When returned to the Terminal prompt after a **BREAK** or an error, InterSystems IRIS keeps track of the location of the command that caused the **BREAK** or error. Later, you can resume execution at the next command simply by entering an argumentless **GOTO** at the Terminal prompt:

Terminal

```
USER 4f0>GOTO
```

By typing a **GOTO** with an argument, you can resume execution at the beginning of another line in the same routine with the break or error, as follows:

Terminal

```
USER 4f0>GOTO label3
```

You can also resume execution at the beginning of a line in a different routine:

Terminal

```
USER 4f0>GOTO label3^rou
```

Alternatively, you may clear the program stack with an argumentless **QUIT** command:

Terminal

```
USER 4f0>QUIT
USER>
```

26.3.5.1 Sample Dialogs

The following routines are used in the examples below:

```
MAIN ; 03 Jan 2019 11:40 AM
 SET x=1,y=6,z=8
 DO ^SUB1 WRITE !,"sum=",sum
 QUIT

SUB1 ; 03 Jan 2019 11:42 AM
 SET sum=x+y+z
 QUIT
```

With **BREAK "L"**, breaking does not occur in the routine SUB1.

Terminal

```
USER>BREAK "L"
USER>DO ^MAIN
SET x=1,y=6,z=8
^
<BREAK>MAIN+1^MAIN
USER 2d0>GOTO
DO ^SUB1 WRITE !,"sum=",sum
^
<BREAK>MAIN+2^MAIN
USER 2d0>GOTO
sum=15
QUIT
^
<BREAK>MAIN+3^MAIN
USER 2d0>GOTO
USER>
```

With **BREAK** "L+", breaking also occurs in the routine SUB1.

Terminal

```
USER>BREAK "L+"
USER>DO ^MAIN
SET x=1,y=6,z=8
^
<BREAK>MAIN+1^MAIN
USER 2d0>GOTO
DO ^SUB1 WRITE !,"sum=",sum
^
<BREAK>MAIN+2^MAIN
USER 2d0>GOTO
SET sum=x+y+z
^
<BREAK>SUB1+1^SUB1
USER 3d0>GOTO
QUIT
^
<BREAK>SUB1+2^SUB1
USER 3d0>GOTO
sum=15
QUIT
^
<BREAK>MAIN+3^MAIN
USER 2d0>GOTO
USER>
```

26.3.6 The NEW Command at the Terminal Prompt

The argumentless NEW command effectively saves all symbols in the symbol table so you can proceed with an empty symbol table. You may find this command particularly valuable after an error or **BREAK**.

To run other routines without disturbing the symbol table, issue an argumentless **NEW** command at the Terminal prompt. The system then:

- Stacks the current frame on the program stack.
- Returns the Terminal prompt for a new stack frame.

For example:

Terminal

```
USER 4d0>NEW
USER 5B1>DO ^%T
3:49 PM
USER 5B1>QUIT 1
USER 4d0>GOTO
```

The 5B1> prompt indicates that the system has stacked the current frame entered through a **BREAK**. The 1 indicates that a **NEW** command has stacked variable information, which you can remove by issuing a **QUIT 1**. When you wish to resume execution, issue a **QUIT 1** to restore the old symbol table, and a **GOTO** to resume execution.

Whenever you use a **NEW** command, parameter passing, or user-defined function, the system places information on the stack indicating that later an explicit or implicit **QUIT** at the current subroutine or XECUTE level should delete certain variables and restore the value of others.

You may find it useful to know if any **NEW** commands, parameter passing, or user-defined functions have been executed (thus stacking some variables), and if so, how far back on the stack this information resides.

26.3.7 The QUIT Command at the Terminal Prompt

From the Terminal prompt you can remove all items from the program stack by entering an argumentless QUIT command:

Terminal

```
USER 4f0>QUIT
USER>
```

To remove only a couple of items from the program stack (for example, to leave a currently executing subroutine and return to a previous **DO** level), use **QUIT** with an integer argument. QUIT 1 removes the last item on the program stack, QUIT 3 removes the last three items, and so forth, as illustrated below:

Terminal

```
9f0>QUIT 3
6d0>
```

26.3.8 InterSystems IRIS Error Messages

InterSystems IRIS displays error messages within angle brackets, as in <ERROR>, followed by a reference to the line that was executing at the time of the error and by the routine. A caret (^) separates the line reference and routine. Also displayed is the intermediate code line with a caret character under the first character of the command executing when the error occurred. For example:

```
SET x=y+3 DO ^ABC
^
<UNDEFINED>label+3^rou
```

This error message indicates an <UNDEFINED> error (that refers to the variable *y*) in line label+3 of routine `rou`. At this point, this message is also the value of the special variable $ZERROR.

26.4 Using %STACK to Display the Stack

You can use the %STACK utility to:

- Display the contents of the process execution stack.
- Display the values of local variables, including values that have been “hidden” with the **NEW** command or through parameter passing.
- Display the values of process state variables, such as $IO and $JOB.

26.4.1 Running %STACK

You execute %STACK by entering the following command:

Terminal

```
USER>DO ^%STACK
```

As shown in this example, the %STACK utility displays the current process stack without variables.

```
Level  Type     Line                   Source
  1   SIGN ON
  2   DO                                 ~DO ^StackTest
  3   NEW ALL/EXCL                       NEW (E)
  4   DO                 TEST+1^StackTest          SET A=1 ~DO TEST1 QUIT  ;level=2
  5   NEW                                NEW A
  6   DO                TEST1+1^StackTest          ~DO TEST2 ;level = 3
  7   ERROR TRAP                         SET $ZTRAP="TrapLabel^StackTest"
  8   XECUTE            TEST2+2^StackTest  ~XECUTE "SET A=$$TEST3()"
  9   $$EXTFUNC                ^StackTest ~SET A=$$TEST3()
 10   PARAMETER                          AA
 11   DIRECT BREAK      TEST3+1^StackTest           ~BREAK
 12   DO                       ^StackTest ~DO ^%STACK
```

Under the current execution stack display, %STACK prompts you for a **Stack Display Action**. You can get help by entering a question mark (?) at this prompt. You can exit %STACK by pressing the **Return** key at this prompt.

26.4.2 Displaying the Process Execution Stack

Depending on what you enter at the **Stack Display Action** prompt, you can display the current process execution stack in four forms:

- Without variables, by entering *F
- With a specific local variable, by entering *V
- With all local variables, by entering *P
- With all local variables, preceded by a list of process state variables, by entering *A

%STACK then displays the **Display on Device** prompt, enabling you to specify where you want this information to go. Press the **Return** key to display this information to the current device.

26.4.2.1 Displaying the Stack without Variables

The process execution stack without variables appears when you first enter the %STACK utility or when you type *F at the **Stack Display Action** prompt.

26.4.2.2 Displaying the Stack with a Specific Variable

Enter *V at the **Stack Display Action** prompt. This will prompt you for the name(s) of the local variable(s) you want to track through the stack. Specify a single variable or a comma-separated list of variables. It returns the names and values of all local variables. In the following example, the variable *e* is being tracked and the display is sent to the Terminal by pressing **Return**

```
Stack Display Action: *V
Now loading variable information ...   2  done.
Variable(s): e
Display on
Device: <RETURN>
```

26.4.2.3 Displaying the Stack with All Defined Variables

Enter *P at the **Stack Display Action** prompt to see the process execution stack together with the current values of all defined local variables.

26.4.2.4 Displaying the Stack with All Variables, including State Variables

Enter *A at the **Stack Display Action** prompt to display all possible reports. Reports are issued in the following order:

- Process state intrinsic variables
- Process execution stack with the names and values of all local variables

26.4.3 Understanding the Stack Display

Each item on the stack is called a *frame*. The following table describes the information provided for each frame.

Table 26–2: %STACK Utility Information

Heading	Description
Level	Identifies the level within the stack. The oldest item on the stack is number 1. Frames without an associated level number share the level that first appears above them.
Type	Identifies the type of frame on the stack, which can be: DIRECT BREAK: A **BREAK** command was encountered that caused a return to direct mode. DIRECT CALLIN: An InterSystems IRIS process was initiated from an application outside of InterSystems IRIS, using the InterSystems IRIS call-in interface. DIRECT ERROR: An error was encountered that caused a return to direct mode. DO: A DO command was executed. ERROR TRAP: If a routine sets $ZTRAP, this frame identifies the location where an error will cause execution to continue. FOR: A FOR command was executed. NEW: A NEW command was executed. If the **NEW** command had arguments, they are shown. SIGN ON: Execution of the InterSystems IRIS process was initiated. XECUTE: An XECUTE command was executed. An $XECUTE function was executed. $$EXTFUNC: A user-defined function was executed.
Line	Identifies the ObjectScript source line associated with the frame, if available, in the format label+offset^routine.
Source	Shows the source code for the line, if it is available. If the source is too long to display in the area provided, horizontal scrolling is available. If the device is line- oriented, the source wraps around and continued lines are preceded with `. . . .`

The following table shows whether level, line, and source values are available for each frame type. A "No" under Level indicates that the level number is not incremented and no level number appears in the display.

Table 26–3: Frame Types and Values Available

Frame Type	Level	Line	Source
DIRECT BREAK	Yes	Yes	Yes
DIRECT CALL IN	Yes	No	No
DIRECT ERROR	Yes	Yes	Yes
DO	Yes	Yes*	Yes
ERROR TRAP	No	No	No, but the new $ZTRAP value is shown.
FOR	No	Yes	Yes
NEW	No	No	Shows the form of the NEW (inclusive or exclusive) and the variables affected.
PARAMETER	No	No	Shows the formal parameter list. If a parameter is passed by reference, shows what other variables point to the same memory location.
SIGN ON	Yes	No	No

Frame Type	Level	Line	Source
XECUTE	Yes	Yes*	Yes
$$EXTFUNC	Yes	Yes*	Yes
* The LINE value is blank if these are invoked from the Terminal prompt.			

26.4.3.1 Moving through %STACK Display

If a %STACK display fills more than one screen, you see the prompt `-- more --` in the bottom left corner of the screen. At the last page, you see the prompt `-- fini --`. Type ? to see key presses you use to maneuver through the %STACK display.

```
- - - Filter Help - - -
<space> Display next page.
<return> Display one more line.
T Return to the beginning of the output.
B Back up one page (or many if arg>1).
R Redraw the current page.
/text Search for \qtext\q after the current page.
A View all the remaining text.
Q Quit.
? Display this screen
# specify an argument for B, L, or W actions.
L set the page length to the current argument.
W set the page width to the current argument.
```

You enter any of the commands listed above whenever you see the `-- more --` or `-- fini --` prompts.

For the B, L and W commands, you enter a numeric argument before the command letter. For instance, enter 2B to move back two pages, or enter 20L to set the page length to 20 lines.

Be sure to set your page length to the number of lines which are actually displayed; otherwise, when you do a page up or down, some lines may not be visible. The default page length is 23.

26.4.3.2 Displaying Variables at Specific Stack Level

To see the variables that exist at a given stack frame level, enter ?# at the `Stack Display Action` prompt, where # is the stack frame level. The following example shows the display if you request the variables at level 1.

```
Stack Display Action: ?1
The following Variables are defined for Stack Level: 1
E
Stack Display Action:
```

You can also display this information using the %SYS.ProcessQuery VariableList class query.

26.4.3.3 Displaying Stack Levels with Variables

You can display the variables defined at all stack levels by entering ?? at the `Stack Display Action` prompt. The following example shows a sample display if you select this action.

```
Stack Display Action: ??
Now loading variable information ... 19
Base Stack Level: 5
A
Base Stack Level: 3
A B C D
Base Stack Level: 1
E
Stack Display Action:
```

26.4.3.4 Displaying Process State Variables

To display the process state variables, such as **$IO**, enter *S at the "Stack Display Action" prompt. You will see these defined variables (Process State Intrinsics) as listed in the following table:

Process State Intrinsics	Documentation
$D =	$DEVICE special variable
$EC = ,M9,	$ECODE special variable
$ES = 4	$ESTACK special variable
$ET =	
$H = 64700,50668	$HOROLOG special variable
$I = |TRM|:|5008	$IO special variable
$J = 5008	$JOB special variable
$K = $c(13)	$KEY special variable
$P = |TRM|:|5008	$PRINCIPAL special variable
$Roles = %All	$ROLES special variable
$S = 268315992	$STORAGE special variable
$T = 0	$TEST special variable
$TL = 0	$TLEVEL special variable
$USERNAME = glenn	$USERNAME special variable
$X = 0	$X special variable
$Y = 17	$Y special variable
$ZA = 0	$ZA special variable
$ZB = $c(13)	$ZB special variable
$ZC = 0	$ZCHILD special variable
$ZE = <DIVIDE>	$ZERROR special variable
$ZJ = 5	$ZJOB special variable
$ZM = RY\Latin1\K\UTF8\	$ZMODE special variable
$ZP = 0	$ZPARENT special variable
$ZR = ^||a	$ZREFERENCE special variable
$ZS = 262144	$ZSTORAGE special variable
$ZT =	$ZTRAP special variable
$ZTS = 64700,68668.58	$ZTIMESTAMP special variable
$ZU(5) = USER	$NAMESPACE
$ZU(12) = c:\intersystems\iris\mgr\	**NormalizeDirectory()**
$ZU(18) = 0	**Undefined()**
$ZU(20) = USER	**UserRoutinePath()**

Process State Intrinsics	Documentation
$ZU(23,1) = 5	
$ZU(34) = 0	
$ZU(39) = USER	**SysRoutinePath()**
$ZU(55) = 0	**LanguageMode()**
$ZU(56,0) = $Id: //iris/2018.1.1/kernel/common/src/emath.c#1 $ 0	
$ZU(56,1) = 1349	
$ZU(61) = 16	
$ZU(61,30,n) = 262160	
$ZU(67,10,$J) = 1	*JobType*
$ZU(67,11,$J) = glenn	*UserName*
$ZU(67,12,$J) = TRM:	*ClientNodeName*
$ZU(67,13,$J) =	*ClientExecutableName*
$ZU(67,14,$J) =	*CSPSessionID*
$ZU(67,15,$J) = 127.0.0.1	*ClientIPAddress*
$ZU(67,4,$J) = 0^0^0	*State*
$ZU(67,5,$J) = %STACK	*Routine*
$ZU(67,6,$J) = USER	*NameSpace*
$ZU(67,7,$J) = |TRM|:|5008	*CurrentDevice*
$ZU(67,8,$J) = 923	*LinesExecuted*
$ZU(67,9,$J) = 46	*GlobalReferences*
$ZU(68,1) = 0	**NullSubscripts()**
$ZU(68,21) = 0	**SynchCommit()**
$ZU(68,25) = 0	
$ZU(68,27) = 1	
$ZU(68,32) = 0	**ZDateNull()**
$ZU(68,34) = 1	**AsynchError()**
$ZU(68,36) = 0	
$ZU(68,40) = 0	**SetZEOF()**
$ZU(68,41) = 1	
$ZU(68,43) = 0	**OldZU5()**
$ZU(68,5) = 1	**BreakMode()**
$ZU(68,6) = 0	
$ZU(68,7) = 0	**RefInKind()**

Process State Intrinsics	Documentation
$ZU(131,0) = MYCOMPUTER	
$ZU(131,1) = MYCOMPUTER:IRIS	
$ZV = IRIS for Windows (x86-64) 2018.1.0 (Build 527U) Tue Feb 20 2018 22:47:10 EST	$ZVERSION special variable

26.4.3.5 Printing the Stack and/or Variables

When you select the following actions, you can choose the output device:

- *P
- *A
- *V after selecting the variables you want to display.

26.5 Other Debugging Tools

There are also other tools available to aid in the debugging process. These include:

- Displaying References to an Object with $SYSTEM.OBJ.ShowReferences
- Error Trap Utilities — %ETN and %ERN

26.5.1 Displaying References to an Object with $SYSTEM.OBJ.ShowReferences

To display all variables in the process symbol table that contain a reference to a given object, use the **ShowReferences(oref)** method of the %SYSTEM.OBJ class. The *oref* is the OREF (object reference) for the given object. For details on OREFs, see OREF Basics.

26.5.2 Error Trap Utilities

The error trap utilities, %ETN and %ERN, help in error analysis by storing variables and recording other pertinent information about an error.

26.5.2.1 %ETN Application Error Trap

You may find it convenient to set the error trap to execute the utility **%ETN** on an application error. This utility saves valuable information about the job at the time of the error, such as the execution stack and the value of variables. This information is saved in the application error log, which you can display with the **%ERN** utility or view in the Management Portal on the **View Application Error Log** page (**System Operation**, **System Logs**, **Application Error Log**).

Use the following code to set the error trap to this utility:

```
SET $ZTRAP="^%ETN"
```

Note: In a procedure, you cannot set *$ZTRAP* to an external routine. Because of this restriction, you cannot use **^%ETN** in procedures (including class methods that are procedures). However, you can set *$ZTRAP* to a local label that calls **%ETN**.

When an error occurs and you call the %ETN utility, you see a message similar to the following message:

```
Error has occurred: <SYNTAX> at 10:30 AM
```

Because **%ETN** ends with a **HALT** command (terminates the process) you may want to set the **%ETN** error trap only if the routine is used in Application Mode. When an error occurs at the Terminal prompt, it may be useful for the error to be displayed on the terminal and go into the debugger prompt to allow for immediate analysis of the error. The following code sets an error trap only if InterSystems IRIS is in Application Mode:

```
SET $ZTRAP=$SELECT($ZJ#2:"",1:"^%ETN")
```

26.5.2.2 %ERN Application Error Report

The %ERN utility examines application errors recorded by the %ETN error trap utility. See Using %ERN to View Application Error Logs.

In the following code, a **ZLOAD** of the routine REPORT is issued to illustrate that by loading all of the variables with `*LOAD` and then loading the routine, you can recreate the state of the job when the error occurred except that the program stack, which records information about **DO**s, etc., is empty.

Terminal

```
USER>DO ^%ERN

For Date: 4/30/2018   3 Errors

Error: ?L

1) "<DIVIDE>zMyTest+2^Sample.MyStuff.1"  at 10:27 am.   $I=|TRM|:|10044   ($X=0  $Y=17)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=262144 ($S=268242904)
               WRITE 5/0

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16 pm. $I=|TRM|:|10044   ($X=0  $Y=57)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=2147483647 ($S=2199023047592)
 SET ^REPORT(%DAT,TYPE)=I

3) <UNDEFINED>zMyTest+2^Sample.MyStuff.1 *undef"  at 10:13 pm.   $I=|TRM|:|12416   ($X=0  $Y=7)
     $J=12416  $ZA=0   $ZB=$c(13)   $ZS=262144 ($S=268279776)
               WRITE undef

Error: 2

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16 pm. $I=|TRM|:|10044   ($X=0  $Y=57)
     $J=10044  $ZA=0   $ZB=$c(13)   $ZS=2147483647 ($S=2199023047592)
 SET ^REPORT(%DAT,TYPE)=I

Variable: %DAT
  %DAT="Apr 30 2018"

Variable: TYPE
  TYPE=""

Variable: *LOAD
USER>ZLOAD REPORT

USER>WRITE

%DAT="Apr 30 2018"
%DS=""
%TG="REPORT+1"
I=88
TYPE=""
XY="SET $X=250 WRITE *27,*91,DY+1,*59,DX+1,*72 SET $X=DX,$Y=DY"
USER>
```

A

(Legacy) Using ^%ETN for Error Logging

An older style of error logging uses the **^%ETN** utility, described here for reference.

The **^%ETN** utility logs an exception to the application error log and then exits. You can invoke **^%ETN** (or one of its entry points) as a utility:

ObjectScript

```
DO ^%ETN
```

Or you can set the $ZTRAP special variable equal to ^%ETN (or one of its entry points):

ObjectScript

```
SET $ZTRAP="^%ETN"
```

You can specify **^%ETN** or one of its entry points:

- **FORE^%ETN** (foreground) logs an exception to the standard application error log, and then exits with a HALT. This invokes a rollback operation. This is the same operation as **^%ETN**.
- BACK^%ETN (background) logs an exception to the standard application error log, and then exits with a QUIT. This does not invoke a rollback operation.
- **LOG^%ETN** logs an exception to the standard application error log, and then exits with a QUIT. This does not invoke a rollback operation. The exception can be a standard %Exception.SystemException, or a user-defined exception.

  To define an exception, set **$ZERROR** to a meaningful value prior to calling **LOG^%ETN**; this value will be used as the Error Message field in the log entry. You can also specify a user-defined exception directly into **LOG^%ETN**: `DO LOG^%ETN("This is my custom exception")`; this value will be used as the Error Message field in the log entry. If you set **$ZERROR** to the null string (`SET $ZERROR=""`) **LOG^%ETN** logs a <LOG ENTRY> error. If you set **$ZERROR** to <INTERRUPT> (`SET $ZERROR="<INTERRUPT>"`) **LOG^%ETN** logs an <INTERRUPT LOG> error.

  LOG^%ETN returns a %List structure with two elements: the $HOROLOG date and the Error Number.

The following example uses the recommended coding practice of immediately copying **$ZERROR** into a variable. **LOG^%ETN** returns a %List value:

ObjectScript

```
SET err=$ZERROR
/* error handling code */
SET rtn = $$LOG^%ETN(err)
WRITE "logged error date: ",$LIST(rtn,1),!
WRITE "logged error number: ",$LIST(rtn,2)
```

Calling **LOG^%ETN** or **BACK^%ETN** automatically increases the available process memory, does the work, and then restores the original $ZSTORAGE value. However, if you call **LOG^%ETN** or **BACK^%ETN** following a <STORE> error, restoring the original **$ZSTORAGE** value might trigger another <STORE> error. For this reason, the system retains the increased available memory when these **^%ETN** entry points are invoked for a <STORE> error.

B (Legacy) Traditional Error Processing

This page describes error processing that uses $ZTRAP, a form of error processing that may be encountered in legacy applications. New applications should use TRY-CATCH instead.

B.1 How Traditional Error Processing Works

For traditional error processing, InterSystems IRIS® data platform enables your application to have an *error handler*. An error handler processes any error that may occur while the application is running. A special variable specifies the ObjectScript commands to be executed when an error occurs. These commands may handle the error directly or may call a routine to handle it.

To set up an error handler, the basic process is:

1. Create one or more routines to perform error processing. Write code to perform error processing. This can be general code for the entire application or specific processing for specific error conditions. This allows you to perform customized error handling for each particular part of an application.
2. Establish one or more error handlers within your application, each using specific appropriate error processing.

If an error occurs and no error handler has been established, the behavior depends on how the InterSystems IRIS session was started:

1. If you signed onto InterSystems IRIS at the Terminal prompt and have not set an error trap, InterSystems IRIS displays an error message on the principal device and returns the Terminal prompt with the program stack intact. The programmer can later resume execution of the program.
2. If you invoked InterSystems IRIS in Application Mode and have not set an error trap, InterSystems IRIS displays an error message on the principal device and executes a **HALT** command.

B.1.1 Internal Error-Trapping Behavior

To get the full benefit of InterSystems IRIS error processing and the scoping issues surrounding the **$ZTRAP** special variable (as well as **$ETRAP**), it is helpful to understand how InterSystems IRIS transfers control from one routine to another.

InterSystems IRIS builds a data structure called a “context frame” each time any of the following occurs:

- A routine calls another routine with a **DO** command. (This kind of frame is also known as a “**DO** frame.”)

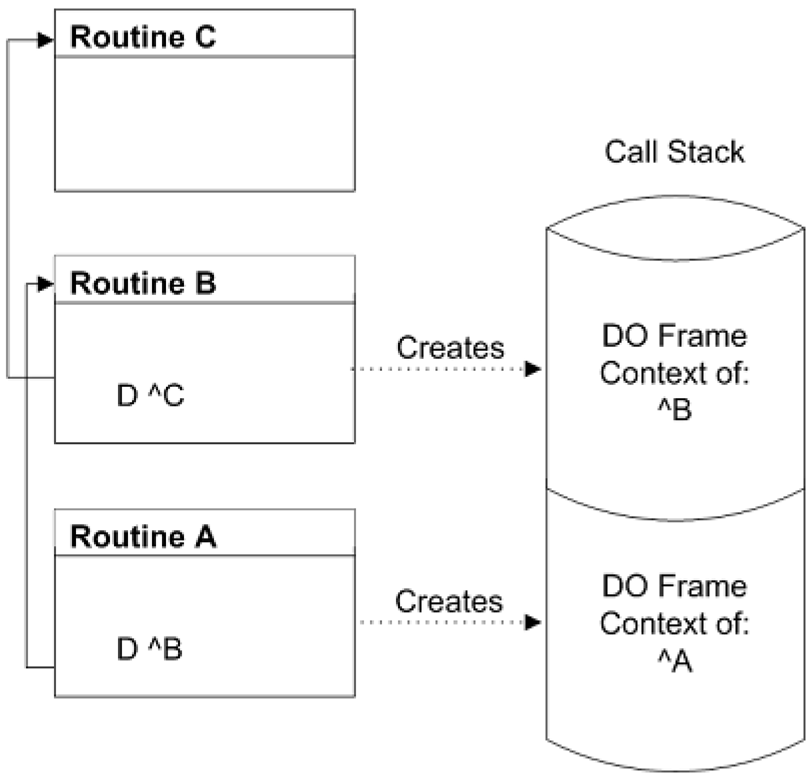
- An **XECUTE** command argument causes ObjectScript code to execute. (This kind of frame is also known as a “**XECUTE** frame.”)
- A user-defined function is executed.

The frame is built on the call stack, one of the private data structures in the address space of your process. InterSystems IRIS stores the following elements in the frame for a routine:

- The value of the **$ZTRAP** special variable (if any)
- The value of the **$ETRAP** special variable (if any)
- The position to return from the subroutine

When routine A calls routine B with `DO ^B`, InterSystems IRIS builds a **DO** frame on the call stack to preserve the context of A. When routine B calls routine C, InterSystems IRIS adds a **DO** frame to the call stack to preserve the context of B, and so forth.

Figure II–1: Frames on a Call Stack



If routine A in the figure above is invoked at the Terminal prompt using the **DO** command, then an extra **DO** frame, not described in the figure, exists at the base of the call stack.

B.1.2 Current Context Level

You can use the following to return information about the current context level:

- The **$STACK** special variable contains the current relative stack level.
- The **$ESTACK** special variable contains the current stack level. It can be initialized to 0 (level zero) at any user-specified point.
- The **$STACK** function returns information about the current context and contexts that have been saved on the call stack

B.1.2.1 The $STACK Special Variable

The **$STACK** special variable contains the number of frames currently saved on the call stack for your process. The **$STACK** value is essentially the context level number (zero based) of the currently executing context. Therefore, when an image is started, but before any commands are processed, the value of **$STACK** is 0.

See the $STACK special variable in the *ObjectScript Reference* for details.

B.1.2.2 The $ESTACK Special Variable

The **$ESTACK** special variable is similar to the **$STACK** special variable, but is more useful in error handling because you can reset it to 0 (and save its previous value) with the **NEW** command. Thus, a process can reset **$ESTACK** in a particular context to mark it as a **$ESTACK** level 0 context. Later, if an error occurs, error handlers can test the value of **$ESTACK** to unwind the call stack back to that context.

See the $ESTACK special variable in the *ObjectScript Reference* for details.

B.1.2.3 The $STACK Function

The **$STACK** function returns information about the current context and contexts that have been saved on the call stack. For each context, the **$STACK** function provides the following information:

- The type of context (**DO**, **XECUTE**, or user-defined function)
- The entry reference and command number of the last command processed in the context
- The source routine line or **XECUTE** string that contains the last command processed in the context
- The **$ECODE** value of any error that occurred in the context (available only during error processing when **$ECODE** is non-null)

When an error occurs, all context information is immediately saved on your process error stack. The context information is then accessible by the **$STACK** function until the value of **$ECODE** is cleared by an error handler. In other words, while the value of **$ECODE** is non-null, the **$STACK** function returns information about a context saved on the error stack rather than an active context at the same specified context level.

See the $STACK function in the *ObjectScript Reference* for details.

When an error occurs and an error stack already exists, InterSystems IRIS records information about the new error at the context level where the error occurred, unless information about another error already exists at that context level on the error stack. In this case, the information is placed at the next level on the error stack (regardless of the information that may already be recorded there).

Therefore, depending on the context level of the new error, the error stack may extend (one or more context levels added) or information at an existing error stack context level may be overwritten to accommodate information about the new error.

Keep in mind that you clear your process error stack by clearing the **$ECODE** special variable.

B.1.3 Error Codes

When an error occurs, InterSystems IRIS sets the **$ZERROR** and **$ECODE** special variables to a value describing the error. The **$ZERROR** and **$ECODE** values are intended for use immediately following an error. Because these values may not be preserved across routine calls, users who wish to preserve a value for later use should copy it to a variable.

Important: Do not use error code values to try to detect error conditions. To *detect* error conditions, use a TRY-CATCH block or $ZTRAP, which are designed for that purpose.

B.1.3.1 $ZERROR Value

InterSystems IRIS sets **$ZERROR** to a string containing:

- The InterSystems IRIS error code, enclosed in angle brackets.
- The label, offset, and routine name where the error occurred.
- (For some errors): Additional information, such as the name of the item that caused the error.

The **AsSystemError()** method of the %Exception.SystemException class returns the same values in the same format as **$ZERROR**.

The following examples show the type of messages to which **$ZERROR** is set when InterSystems IRIS encounters an error. In the following example, the undefined local variable *abc* is invoked at line offset 2 from label PrintResult of routine MyTest. **$ZERROR** contains:

```
<UNDEFINED>PrintResult+2^MyTest *abc
```

The following error occurred when a non-existent class is invoked at line offset 3:

```
<CLASS DOES NOT EXIST>PrintResult+3^MyTest *%SYSTEM.XXQL
```

The following error occurred when a non-existent method of an existing class is invoked at line offset 4:

```
<METHOD DOES NOT EXIST>PrintResult+4^MyTest *BadMethod,%SYSTEM.SQL
```

You can also explicitly set the special variable **$ZERROR** as any string up to 128 characters; for example:

ObjectScript

```
 SET $ZERROR="Any String"
```

The **$ZERROR** value is intended for use immediately following an error. Because a **$ZERROR** value may not be preserved across routine calls, users that wish to preserve a **$ZERROR** value for later use should copy it to a variable. It is strongly recommended that users set **$ZERROR** to the null string ("") immediately after use. See the $ZERROR special variable in the *ObjectScript Reference* for details. For further information on handling **$ZERROR** errors, refer to the %SYSTEM.Error class methods in the *InterSystems Class Reference*.

B.1.3.2 $ECODE Value

When an error occurs, InterSystems IRIS sets **$ECODE** to the value of a comma-surrounded string containing the ANSI Standard error code that corresponds to the error. For example, when you make a reference to an undefined global variable, InterSystems IRIS sets **$ECODE** set to the following string:

```
,M7,
```

If the error has no corresponding ANSI Standard error code, InterSystems IRIS sets **$ECODE** to the value of a comma-surrounded string containing the InterSystems IRIS error code preceded by the letter Z. For example, if a process has exhausted its symbol table space, InterSystems IRIS places the error code <STORE> in the **$ZERROR** special variable and sets **$ECODE** to this string:

```
,ZSTORE,
```

After an error occurs, your error handlers can test for specific error codes by examining the value of the **$ZERROR** special variable or the **$ECODE** special variable.

Note: Error handlers should examine **$ZERROR** rather than **$ECODE** special variable for specific errors.

See the $ECODE special variable in the *ObjectScript Reference* for details.

B.2 Handling Errors with $ZTRAP

To handle errors with **$ZTRAP**, you set the **$ZTRAP** special variable to a *location*, specified as a quoted string. You set the **$ZTRAP** special variable to an entry reference that specifies the *location* to which control is to be transferred when an error occurs. You then write **$ZTRAP** code at that location.

When you set **$ZTRAP** to a non-empty value, it takes precedence over any existing **$ETRAP** error handler. InterSystems IRIS implicitly performs a `NEW $ETRAP` command and sets **$ETRAP** equal to "".

B.2.1 Setting $ZTRAP in a Procedure

Within a procedure, you can only set the **$ZTRAP** special variable to a line label (private label) within that procedure. You cannot set **$ZTRAP** to any external routine from within a procedure block.

When displaying the **$ZTRAP** value, InterSystems IRIS does not return the name of the private label. Instead, it returns the offset from the top of the procedure where that private label is located.

For further details see the $ZTRAP special variable in the *ObjectScript Reference*.

B.2.2 Setting $ZTRAP in a Routine

Within a routine, you can set the **$ZTRAP** special variable to a label in the current routine, to an external routine, or to a label within an external routine. You can only reference an external routine if the routine is not procedure block code. The following example establishes LogErr^ErrRou as the error handler. When an error occurs, InterSystems IRIS executes the code found at the LogErr label within the ^ErrRou routine:

ObjectScript

```
SET $ZTRAP="LogErr^ErrRou"
```

When displaying the **$ZTRAP** value, InterSystems IRIS displays the label name and (when appropriate) the routine name.

A label name must be unique within its first 31 characters. Label names and routine names are case-sensitive.

Within a routine, **$ZTRAP** has three forms:

- SET $ZTRAP="*location*"
- SET $ZTRAP="**location*" which executes in the context in which the error occurred that invoked it.
- SET $ZTRAP="^%ETN" which executes the system-supplied error routine **^%ETN** in the context in which the error occurred that invoked it. You cannot execute ^%ETN (or any external routine) from a procedure block. Either specify the code is [Not ProcedureBlock], or use a routine such as the following, which invokes the **^%ETN** entry point BACK^%ETN:

```
ClassMethod MyTest() as %Status
  {
  SET $ZTRAP="Error"
  SET ans = 5/0      /* divide-by-zero error */
  WRITE "Exiting ##class(User.A).MyTest()",!
  QUIT ans
Error
  SET err=$ZERROR
  SET $ZTRAP=""
  DO BACK^%ETN
  QUIT $$$ERROR($$$CacheError,err)
  }
```

For more information on **^%ETN** and its entry points, see (Legacy) Using ^%ETN for Error Logging. For details on its use with **$ZTRAP**, see SET $ZTRAP=^%ETN.

For further details see the $ZTRAP special variable in the *ObjectScript Reference*.

B.2.3 Writing $ZTRAP Code

The *location* that **$ZTRAP** points to can perform a variety of operations to display, log, and/or correct an error. Regardless of what error handling operations you wish to perform, the **$ZTRAP** code should begin by performing two tasks:

- Set **$ZTRAP** to another value, either the *location* of an error handler, or the empty string (""). (You must use **SET**, because you cannot **KILL $ZTRAP**.) This is done because if another error occurs during error handling, that error would invoke the current **$ZTRAP** error handler. If the current error handler is the error handler you are in, this would result in an infinite loop.
- Set a variable to **$ZERROR**. If you wish to reference a **$ZERROR** value later in your code, refer to this variable, not **$ZERROR** itself. This is done because **$ZERROR** contains the most-recent error, and a **$ZERROR** value may not be preserved across routine calls, including internal routine calls. If another error occurs during error handling, the **$ZERROR** value would be overwritten by that new error.

  It is strongly recommended that users set **$ZERROR** to the null string ("") immediately after use.

The following example shows these essential **$ZTRAP** code statements:

ObjectScript

```
MyErrHandler
  SET $ZTRAP=""
  SET err=$ZERROR
  /* error handling code
     using err as the error
     to be handled */
```

B.2.4 Using $ZTRAP

Each routine in an application can establish its own **$ZTRAP** error handler by setting **$ZTRAP**. When an error trap occurs, InterSystems IRIS takes the following steps:

1. Sets the special variable **$ZERROR** to an error message.
2. Resets the program stack to the state it was in when the error trap was set (when the `SET $ZTRAP=` was executed). In other words, the system removes all entries on the stack until it reaches the point at which the error trap was set. (The program stack is not reset if **$ZTRAP** was set to a string beginning with an asterisk (*).)
3. Resumes the program at the location specified by the value of **$ZTRAP**. The value of **$ZTRAP** remains the same.

   Note: You can explicitly set the variable **$ZERROR** as any string up to 128 characters. Usually you would set **$ZERROR** to a null string, but you can set **$ZERROR** to a value.

B.2.5 Unstacking NEW Commands With Error Traps

When an error trap occurs and the program stack entries are removed, InterSystems IRIS also removes all stacked **NEW** commands back to the subroutine level containing the `SET $ZTRAP=`. However, all **NEW** commands executed at that subroutine level remain, regardless of whether they were added to the stack before or after **$ZTRAP** was set.

For example:

ObjectScript

```
Main
  SET A=1,B=2,C=3,D=4,E=5,F=6
  NEW A,B
  SET $ZTRAP="ErrSub"
  NEW C,D
  DO Sub1
  RETURN
Sub1()
  NEW E,F
  WRITE 6/0    // Error: division by zero
  RETURN
ErrSub()
  WRITE !,"Error is: ",$ZERROR
  WRITE
  RETURN
```

When the error in Sub1 activates the error trap, the former values of E and F stacked in Sub1 are removed, but A, B, C, and D remain stacked.

B.2.6 $ZTRAP Flow of Control Options

After a **$ZTRAP** error handler has been invoked to handle an error and has performed any cleanup or error logging operations, the error handler has three flow control options:

- Handle the error and continue the application.
- Pass control to another error handler
- Terminate the application

B.2.6.1 Continuing the Application

After a **$ZTRAP** error handler has handled an error, you can continue the application by issuing a **GOTO**. You do not have to clear the values of the **$ZERROR** or **$ECODE** special variables to continue normal application processing. However, you should clear **$ZTRAP** (by setting it to the empty string) to avoid a possible infinite error handling loop if another error occurs. See "Handling Errors in an Error Handler" for more information.

After completing error processing, your **$ZTRAP** error handler can use the **GOTO** command to transfer control to a pre-determined restart or continuation point in your application to resume normal application processing.

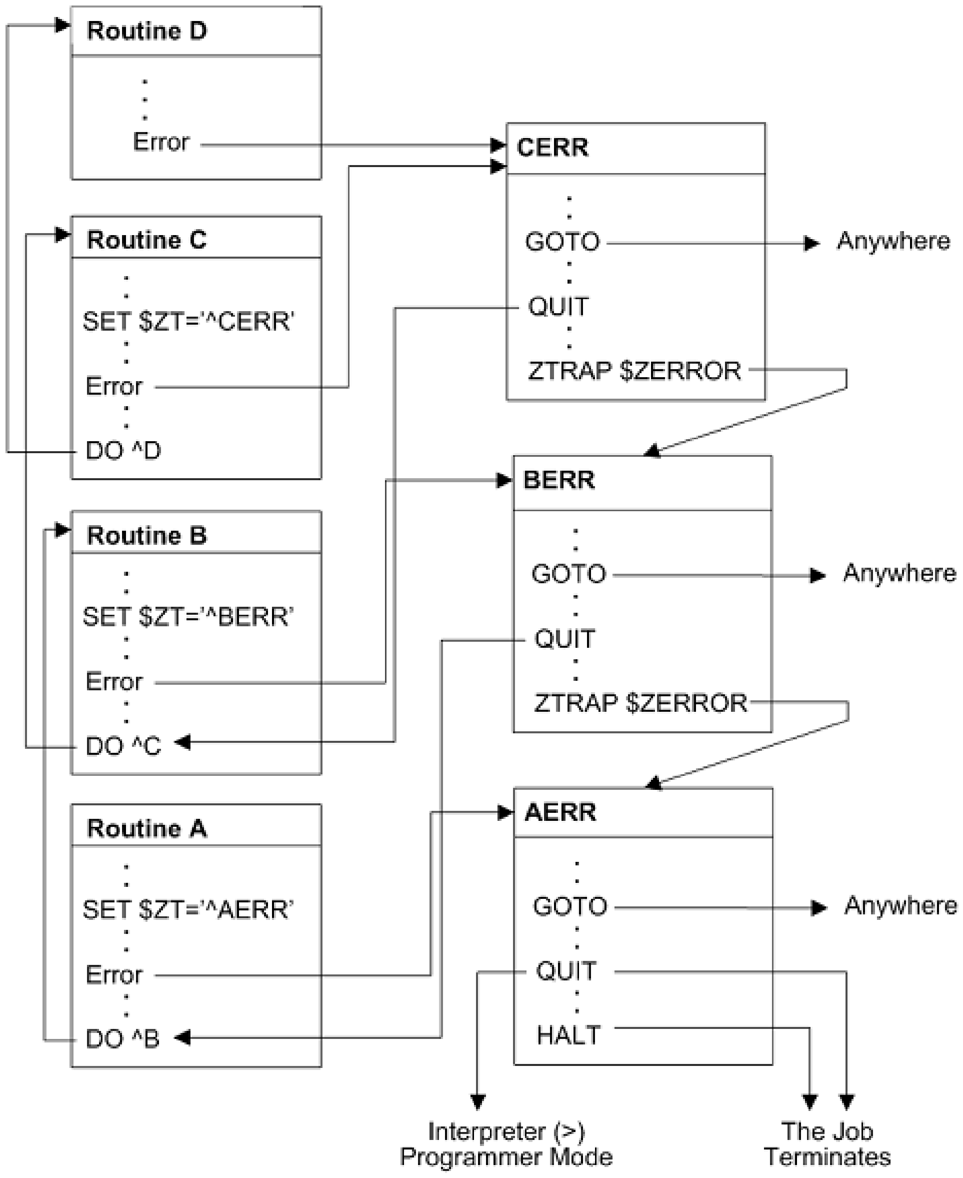
When an error handler has handled an error, the **$ZERROR** special variable is set to a value. This value is not necessarily cleared when the error handler completes. Some routines reset **$ZERROR** to the null string. The **$ZERROR** value is overwritten when the next error occurs that invokes an error handler. For this reason, the **$ZERROR** value should only be accessed within the context of an error handler. If you wish to preserve this value, copy it to a variable and reference that variable, not **$ZERROR** itself. Accessing **$ZERROR** in any other context does not produce reliable results.

B.2.6.2 Passing Control to Another Error Handler

If the error condition cannot be corrected by a **$ZTRAP** error handler, you can use a special form of the **ZTRAP** command to transfer control to another error handler. The command `ZTRAP $ZERROR` re-signals the error condition and causes InterSystems IRIS to unwind the call stack to the next call stack level with an error handler. After InterSystems IRIS has unwound the call stack to the level of the next error handler, processing continues in that error handler. The next error handler may have been set by either a **$ZTRAP** or a **$ETRAP**.

The following figure shows the flow of control in **$ZTRAP** error handling routines.

Figure II–2: $ZTRAP Error Handlers



B.3 Handling Errors with $ETRAP

When an error trap occurs and you have set **$ETRAP**, InterSystems IRIS takes the following steps:

1. Sets the values of **$ECODE** and **$ZERROR**.
2. Processes the commands that are the value of **$ETRAP**.

By default, each **DO**, **XECUTE**, or user-defined function context inherits the **$ETRAP** error handler of the frame that invoked it. This means that the designated **$ETRAP** error handler at any context level is the last defined **$ETRAP**, even if that definition was made several stack levels down from the current level.

B.3.1 $ETRAP Error Handlers

The **$ETRAP** special variable can contain one or more ObjectScript commands that are executed when an error occurs. Use the **SET** command to set **$ETRAP** to a string that contains one or more InterSystems IRIS commands that transfer control to an error-handling routine. This example transfers control to the LogError code label (which is part of the routine ErrRoutine):

ObjectScript

```
SET $ETRAP="DO LogError^ErrRoutine"
```

The commands in the **$ETRAP** special variable are always followed by an implicit **QUIT** command. The implicit **QUIT** command quits with a null string argument when the **$ETRAP** error handler is invoked in a user-defined function context where a **QUIT** with arguments is required.

$ETRAP has a global scope. This means that setting **$ETRAP** should usually be preceded by `NEW $ETRAP`. Otherwise, if the value of **$ETRAP** is set in the current context, then, after passing beyond the scope of that context, the value stored in **$ETRAP** is still present while control is in a higher-level context. Thus, if you do not specify the `NEW $ETRAP`, then **$ETRAP** could be executed at an unexpected time when the context that set that it no longer exists.

See the $ETRAP special variable in the *ObjectScript Reference* for details.

B.3.2 Context-specific $ETRAP Error Handlers

Any context can establish its own **$ETRAP** error handler by taking the following steps:

1. Use the **NEW** command to create a new copy of **$ETRAP**.
2. Set **$ETRAP** to a new value.

If a routine sets **$ETRAP** without first creating a new copy of **$ETRAP**, a new **$ETRAP** error handler is established for the current context, the context that invoked it, and possibly other contexts that have been saved on the call stack. Therefore InterSystems recommends that you create a new copy of **$ETRAP** before you set it.

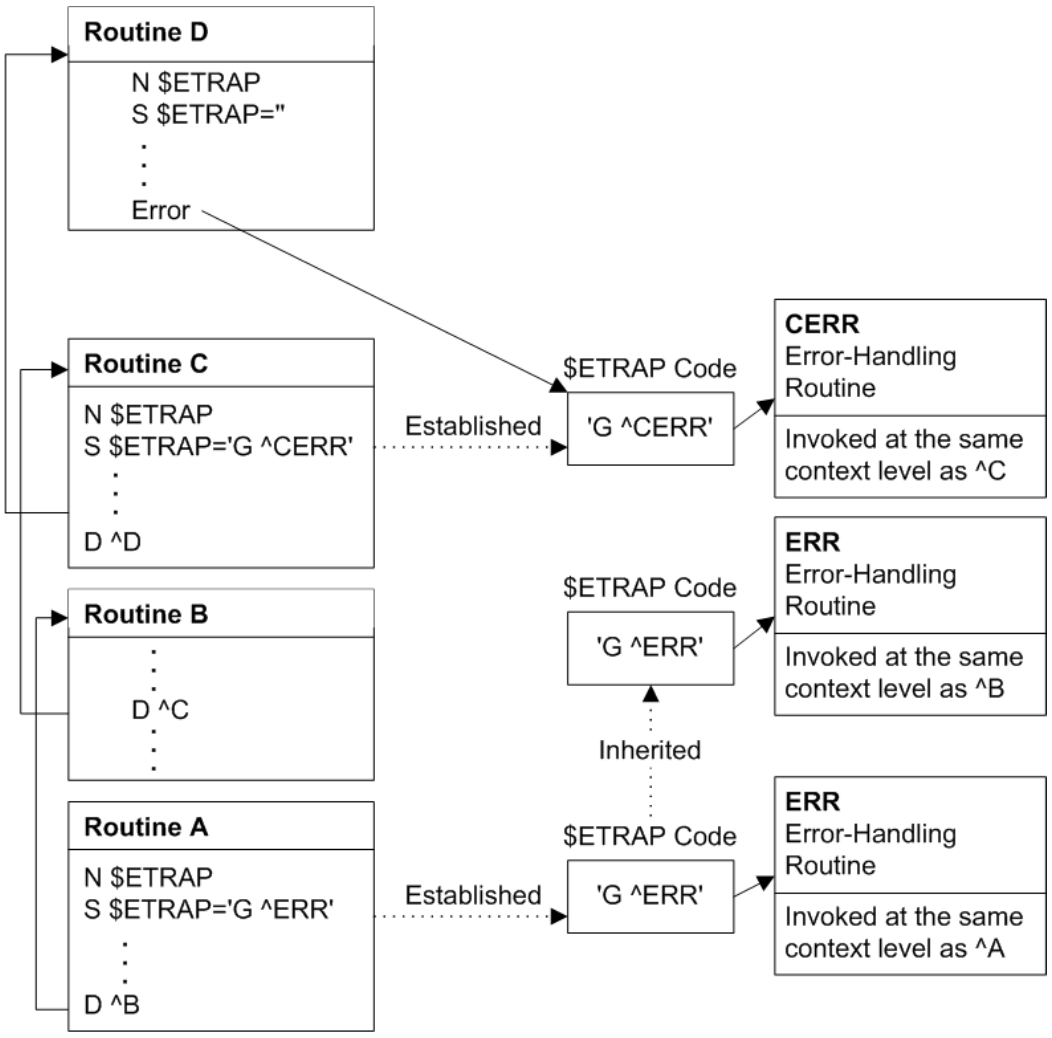
Keep in mind that creating a new copy of **$ETRAP** does not clear **$ETRAP**. The value of **$ETRAP** remains unchanged by the **NEW** command.

The figure below shows the sequence of **$ETRAP** assignments that create the stack of **$ETRAP** error handlers. As the figure shows:

- Routine A creates a new copy of **$ETRAP**, sets it to “GOTO ^ERR”, and contains the **DO** command to call routine B.
- Routine B does nothing with **$ETRAP** (thereby inheriting the **$ETRAP** error handler of Routine A) and contains the **DO** command to call routine C.
- Routine C creates a new copy of **$ETRAP**, sets it to “GOTO ^CERR”, and contains the **DO** command to call routine D.
- Routine D creates a new copy of **$ETRAP** and then clears it, leaving no **$ETRAP** error handler for its context.

If an error occurs in routine D (a context in which no **$ETRAP** error handler is defined), InterSystems IRIS removes the **DO** frame for routine D from the call stack and transfers control to the *$ETRAP* error handler of Routine C. The **$ETRAP** error handler of Routine C, in turn, dispatches to **^CERR** to process the error. If an error occurs in Routine C, InterSystems IRIS transfers control to the **$ETRAP** error handler of Routine C, but does not unwind the stack because the error occurs in a context where a **$ETRAP** error handler is defined.

Figure II–3: $ETRAP Error Handlers



B.3.3 $ETRAP Flow of Control Options

When the **$ETRAP** error handler has been invoked to handle an error and perform any cleanup or error-logging operations, it has the following flow-of-control options:

- Handle the error and continue the application.
- Pass control to another error handler.
- Terminate the application.

B.3.3.1 Handling the Error and Continuing the Application

When a **$ETRAP** error handler is called to handle an error, InterSystems IRIS considers the error condition active until the error condition is dismissed. You dismiss the error condition by setting the **$ECODE** special variable to the null string:

ObjectScript

```
SET $ECODE=""
```

Clearing **$ECODE** also clears the error stack for your process.

Typically, you use the **GOTO** command to transfer control to a predetermined restart or continuation point in your application after the error condition is dismissed. In some cases, you may find it more convenient to quit back to the previous context level after dismissing the error condition.

B.3.3.2 Passing Control to Another Error Handler

If the error condition is not dismissed, InterSystems IRIS passes control to another error handler on the call stack when a **QUIT** command terminates the context at which the **$ETRAP** error handler was invoked. Therefore, you pass control to a previous level error handler by performing a **QUIT** from a **$ETRAP** context without clearing **$ECODE**.

If routine D, called from routine C, contains an error that transfers control to ^CERR, the **QUIT** command in ^CERR that is not preceded by setting **$ECODE** to "" (the empty string) transfers control to the **$ETRAP** error handler at the previous context level. In contrast, if the error condition is dismissed by clearing **$ECODE**, a **QUIT** from **^CERR** transfers control to the statement in routine B that follows the `DO ^C` command.

B.3.3.3 Terminating the Application

If no previous level error handlers exist on the call stack and a **$ETRAP** error handler performs a **QUIT** without dismissing the error condition, the application is terminated. In Application Mode, InterSystems IRIS is then run down and control is passed to the operating system. The Terminal prompt then appears.

Keep in mind that you use the **QUIT** command to terminate a **$ETRAP** error handler context whether or not the error condition is dismissed. Because the same **$ETRAP** error handler can be invoked at context levels that require an argumentless **QUIT** and at context levels (user-defined function contexts) that require a **QUIT** with arguments, the **$QUIT** special variable is provided to indicate the **QUIT** command form required at a particular context level.

The **$QUIT** special variable returns 1 (one) for contexts that require a **QUIT** with arguments and returns 0 (zero) for contexts that require an argumentless **QUIT**.

A **$ETRAP** error handler can use **$QUIT** to provide for either circumstance as follows:

ObjectScript

```
Quit:$QUIT "" Quit
```

When appropriate, a **$ETRAP** error handler can terminate the application using the **HALT** command.

B.4 Handling Errors in an Error Handler

When an error occurs in an error handler, the flow of execution depends on the type of error handler that is currently executing.

B.4.1 Errors in a $ZTRAP Error Handler

If the new error occurs in a **$ZTRAP** error handler, InterSystems IRIS passes control to the first error handler it encounters, unwinding the call stack only if necessary. Therefore, if the **$ZTRAP** error does not clear **$ZTRAP** at the current stack level and another error subsequently occurs in the error handler, the **$ZTRAP** handler is invoked again at the same context level, causing an infinite loop. To avoid this, Set **$ZTRAP** to another value at the beginning of the error handler.

B.4.2 Errors in a $ETRAP Error Handler

If the new error occurs in a **$ETRAP** error handler, InterSystems IRIS unwinds the call stack until the context level at which the **$ETRAP** error handler was invoked has been removed. InterSystems IRIS then passes control to the next error handler (if any) on the call stack.

B.4.3 Error Information in the $ZERROR and $ECODE Special Variables

If another error occurs during the handling of the original error, information about the second error replaces the information about the original error in the **$ZERROR** special variable. However, InterSystems IRIS appends the new information to the **$ECODE** special variable. Depending on the context level of the second error, InterSystems IRIS may append the new information to the process error stack as well.

If the existing value of the **$ECODE** special variable is non-null, InterSystems IRIS appends the code for the new error to the current **$ECODE** value as a new comma piece. Error codes accrue in the **$ECODE** special variable until either of the following occurs:

- You explicitly clear **$ECODE**, for example:

  ObjectScript

  ```
   SET $ECODE = ""
  ```

- The length of **$ECODE** exceeds the maximum string length.

Then, the next new error code replaces the current list of error codes in **$ECODE**.

When an error occurs and an error stack already exists, InterSystems IRIS records information about the new error at the context level where the error occurred, unless information about another error already exists at that context level on the error stack. In this case, the information is placed at the next level on the error stack (regardless of the information that may already be recorded there).

Therefore, depending on the context level of the new error, the error stack may extend (one or more context levels added) or information at an existing error stack context level may be overwritten to accommodate information about the new error.

Keep in mind that you clear your process error stack by clearing the **$ECODE** special variable.

See the $ECODE and $ZERROR special variables in the *ObjectScript Reference* for details. For further information on handling **$ZERROR** errors, refer to the %SYSTEM.Error class methods in the *InterSystems Class Reference*.

B.5 Forcing an Error

You set the **$ECODE** special variable or use the **ZTRAP** command to cause an error to occur under controlled circumstances.

B.5.1 Setting $ECODE

You can set the **$ECODE** special variable to any non-null string to cause an error to occur. When your routine sets **$ECODE** to a non-null string, InterSystems IRIS sets **$ECODE** to the specified string and then generates an error condition. The **$ZERROR** special variable in this circumstance is set with the following error text:

```
<ECODETRAP>
```

Control then passes to error handlers as it does for normal application-level errors.

You can add logic to your error handlers to check for errors caused by setting **$ECODE**. Your error handler can check **$ZERROR** for an <ECODETRAP> error (for example, “$ZE["ECODETRAP"”) or your error handler can check **$ECODE** for a particular string value that you choose.

B.5.2 Creating Application-Specific Errors

Keep in mind that the ANSI Standard format for **$ECODE** is a comma-surrounded list of one or more error codes:

- Errors prefixed with “Z” are implementation-specific errors
- Errors prefixed with “U” are application-specific errors

You can create your own error codes following the ANSI Standard by having the error handler set **$ECODE** to the appropriate error message prefixed with a “U”.

ObjectScript

```
  SET $ECODE=",Upassword expired,"
```

B.6 Processing Errors at the Terminal Prompt

When you generate an error after you sign onto InterSystems IRIS at the Terminal prompt with no error handler set, InterSystems IRIS takes the following steps when an error occurs in a line of code you enter:

1. InterSystems IRIS displays an error message on the process’s principal device.
2. The process breaks at the call stack level where the error occurred.
3. The process returns the Terminal prompt.

B.6.1 Understanding Error Message Formats

As an error message, InterSystems IRIS displays three lines:

1. The entire line of source code in which the error occurred.
2. Below the source code line, a caret (^) points to the command that caused the error.
3. A line containing the contents of $ZERROR.

In the following Terminal prompt example, the second **SET** command has an undefined local variable error:

Terminal

```
USER>WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
hello

WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                              ^
<UNDEFINED> *zzz
USER>
```

In the following example, the same line of code is in a program named `mytest` executed from the Terminal prompt:

Terminal

```
USER>DO ^mytest
hello

  WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                                ^
<UNDEFINED>WriteOut+2^mytest *zzz
USER 2d0>
```

In this case, **$ZERROR** indicates that the error occurred in `mytest` at an offset of 2 lines from the a label named `WriteOut`. Note that the prompt has changed, indicating that a new program stack level has been initiated.

B.6.2 Understanding the Terminal Prompt

By default, the Terminal prompt specifies the current namespace. If one or more transactions are open, it also includes the $TLEVEL transaction level count. This default prompt can be configured with different contents, as described in the ZNSPACE command documentation. The following examples show the defaults:

Terminal

```
USER>
```

Terminal

```
TL1:USER>
```

If an error occurs during the execution of a routine, the system saves the current program stack and initiates a new stack frame. An extended prompt appears, such as:

Terminal

```
USER 2d0>
```

This extended prompt indicates that there are two entries on the program stack, the last of which is an invoking of **DO** (as indicated by the “d”). Note that this error placed two entries on the program stack. The next **DO** execution error would result in the prompt:

Terminal

```
USER 4d0>
```

For a more detailed explanation, see Terminal Prompt Shows Program Stack Information.

B.6.3 Recovering from the Error

You can then take any of the following steps:

- Issue commands from the Terminal prompt
- View and modify your variables and global data
- Edit the routine containing the error or any other routine
- Execute other routines

Any of these steps can even cause additional errors.

After you have taken these steps, your most likely course is to either resume execution or to delete all or part of the program stack.

B.6.3.1 Resuming Execution at the Next Sequential Command

You can resume execution at the next command after the command that caused the error by entering an argumentless **GOTO** from the Terminal prompt:

Terminal

```
USER>DO ^mytest
hello

  WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                                ^
<UNDEFINED>WriteOut+2^mytest *zzz
USER 2d0>GOTO
world

USER>
```

B.6.3.2 Resuming Execution at Another Line

You can resume execution at another line by issuing a **GOTO** with a label argument from the Terminal prompt:

Terminal

```
USER 2d0>GOTO ErrSect
```

B.6.3.3 Deleting the Program Stack

You can delete the entire program stack by issuing an argumentless **QUIT** command from the Terminal prompt:

Terminal

```
USER 4d0>QUIT
USER>
```

B.6.3.4 Deleting Part of the Program Stack

You can issue **QUIT *n*** with an integer argument from the Terminal prompt to delete the last (or last several) program stack entry:

Terminal

```
USER 8d0>QUIT 1

USER 7E0>QUIT 3

USER 4d0>QUIT 1

USER 3E0>QUIT 1

USER 2d0>QUIT 1

USER 1S0>QUIT 1

USER>
```

Note that in this example because the program error created two program stack entries, you must be on a “d” stack entry to resume execution by issuing a **GOTO**. Depending on what else has occurred, a “d” stack entry may be even-numbered (`USER 2d0>`) or odd-numbered (`USER 3d0>`).

By using **NEW $ESTACK**, you can quit to a specified program stack level:

Terminal

```
USER 4d0>NEW $ESTACK

USER 5E1>
/* more errors create more stack frames */

USER 11d7>QUIT $ESTACK

USER 4d0>
```

Note that the **NEW $ESTACK** command adds one entry to the program stack.

C
Legacy Forms of Subroutines

A routine can contain multiple subroutines, using the term *subroutine* in a generic sense. InterSystems recommends that in any new routines you may create, you define procedures. In existing code, you may see subroutines of other forms. This page describes these other forms and explains how to invoke them, if needed.

C.1 Recognizing Legacy Forms

Legacy forms of subroutines use labels but are not enclosed within curly braces. The following list shows the possible syntaxes with their formal names, for reference purposes. In all cases, *label* is the identifier for the unit of code, *args* is the argument list, and the optional *scopekeyword* is either `Public` or `Private`.

subroutine

Formally, a true subroutine (as opposed to a subroutine in a generic sense) is a unit of code of the following form:

```
label(args) scopekeyword
      //implementation
  QUIT
```

See Subroutines.

extrinsic function

Formally, a true extrinsic function (as opposed to a system-defined ObjectScript intrinsic function) is a unit of code of the following form:

```
label(args) scopekeyword
      //implementation
  QUIT optionalreturnvalue
```

A function returns a value, unlike a subroutine. See Functions.

In these legacy forms, variables defined within them are available after the subroutine or function finishes execution. Consequently, these legacy forms use different techniques to manage variable scope—specifically the NEW and KILL commands.

C.2 Subroutines

C.2.1 Syntax

Subroutine syntax:

```
label [ ( param [ = default  ][ , ...] ) ]
   code
   QUIT
```

Invoking syntax:

```
DO label [ ( param [ , ...]  ) ]
```

or

```
GOTO label
```

Argument	Description
label	The name of the subroutine. A standard label. It must start in column one. The parameter parentheses following the label are optional. If specified, the subroutine cannot be invoked using a **GOTO** call. Parameter parentheses prevent code execution from “falling through” into a subroutine from the execution of the code that immediately precedes it. When InterSystems IRIS encounters a label with parameter parentheses (even if they are empty) it performs an implicit **QUIT**, ending execution rather than continuing to the next line in the routine.
param	The parameter value(s) passed from the calling program to the subroutine. A subroutine invoked using the **GOTO** command cannot have *param* values, and must not have parameter parentheses. A subroutine invoked using the **DO** command may or may not have *param* values. If there are no *param* values, empty parameter parentheses may be specified or omitted. Specify a *param* variable for each parameter expected by the subroutine. The expected parameters are known as the *formal parameter list.*. There may be none, one, or more than one *param*. Multiple *param* values are separated by commas. InterSystems IRIS automatically invokes **NEW** on the referenced *param* variables. Parameters may be passed by value or by reference.
default	An optional default value for the *param* preceding it. You can either provide or omit a default value for each parameter. A default value is applied when no actual parameter is provided for that formal parameter, or when an actual parameter is passed by reference and the local variable in question does not have a value. This default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string ("") as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, InterSystems IRIS issues a <PARAMETER> error.

Argument	Description
code	A block of code. This block of code is normally accessed by invoking the label. However, it can also be entered (or reentered) by calling another label within the code block or issuing a label + offset **GOTO** command. A block of code can contain nested calls to other subroutines, functions, or procedures. It is recommended that such nested calls be performed using **DO** commands or function calls, rather than a linked series of **GOTO** commands. This block of code is normally exited by an explicit **QUIT** command; this **QUIT** command is not always required, but is a recommended coding practice. You can also exit a subroutine by using a **GOTO** to an external label.

C.2.2 Description

A subroutine is a block of code identified by a label found in the first column position of the first line of the subroutine. Execution of a subroutine most commonly completes by encountering an explicit **QUIT** statement.

A subroutine is invoked by either the **DO** command or the **GOTO** command.

- A **DO** command executes a subroutine and then resumes execution of the calling routine. Thus, when InterSystems IRIS encounters a **QUIT** command in the subroutine, it returns to the calling routine to execute the next line following the **DO** command.
- A **GOTO** command executes a subroutine but does not return control to the calling program. When InterSystems IRIS encounters a **QUIT** command in the subroutine, execution ceases.

You can pass parameters to a subroutine invoked by the **DO** command; you cannot pass parameters to a subroutine invoked by the **GOTO** command. You can pass parameters by value or by reference. See Passing Arguments.

The same variables are available to a subroutine and its calling routine.

A subroutine does not return a value.

C.3 Functions

A function, by default and recommendation, is a procedure. You can, however, define a function that is not a procedure. This section describes such functions.

C.3.1 Syntax

Non-procedure function syntax:

```
label ( [param [ = default  ]] [ , ...] )
   code
   QUIT expression
```

Invoking syntax:

```
command $$label([param[ ,...]])
```

or

```
DO label([param[ ,...]])
```

Argument	Description
label	The name of the function. A standard label. It must start in column one. The parameter parentheses following the label are mandatory.
param	A variable for each parameter expected by the function. The expected parameters are known as the *formal parameter list* . There may be none, one, or more than one *param*. Multiple *param* values are separated by commas. InterSystems IRIS automatically invokes **NEW** for the referenced *param* variables. Parameters may be passed by value or by reference.
default	An optional default value for the *param* preceding it. You can either provide or omit a default value for each parameter. A default value is applied when no actual parameter is provided for that formal parameter, or when an actual parameter is passed by reference and the local variable in question does not have a value. This default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string () as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, InterSystems IRIS issues a <PARAMETER> error.
code	A block of code. This block of code can contain nested calls to other functions, subroutines, or procedures. Such nested calls must be performed using **DO** commands or function calls. You cannot exit a function's code block by using a **GOTO** command. This block of code can only be exited by an explicit **QUIT** command with an expression.
expression	The function's return value, specified using any valid ObjectScript expression. The **QUIT** command with expression is a mandatory part of a user-defined function. The value that results from expression is returned to the point of invocation as the result of the function.

C.3.2 Description

User-defined functions are described in this section. Calls to user-defined functions are identified by a $$ prefix. (A user-defined function is also known as an *extrinsic function*.)

User-defined functions allow you to add functions to those supplied by InterSystems IRIS. Typically, you use a function to implement a generalized operation that can be invoked from any number of programs.

A function is always called from within an ObjectScript command. It is evaluated as an expression and returns a single value to the invoking command. For example:

ObjectScript

```
SET x=$$myfunc()
```

C.4 Legacy Code and Labels

A procedure is defined by a label and curly braces, which encapsulate the implementation. In contrast, for the legacy forms of subroutines shown on this page, there is no automatic encapsulation of the code. That is, a label provides an entry point, but it does not define an encapsulated unit of code. This means that once the labelled code executes, execution continues into the next labelled unit of code unless execution is stopped or redirected elsewhere. There are three ways to stop execution of a unit of code:

- Execution encounters a QUIT or RETURN.
- Execution encounters the closing curly brace (“}”) of a TRY. When this occurs, execution continues with the next line of code following the associated **CATCH** block.
- Execution encounters the next procedure block (a label with parameter parentheses). Execution stops when encountering a label line with parentheses, even if there are no parameters within the parentheses.

In the following example, code execution continues from the code under label0 to that under label1:

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",! }
  ELSE {DO label1
           WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine0",!
label1
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine1",!
```

In the following example, the labeled code sections end with either a QUIT or RETURN command. This causes execution to stop. Note that **RETURN** always stops execution, **QUIT** stops execution of the current context:

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",! }
  ELSE {DO label1
           WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1
      QUIT }
  WRITE "Quit the FOR loop, not the routine",!
  WRITE "At the end of Routine0",!
  QUIT
  WRITE "This should never print"
label1
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine1",!
  RETURN
  WRITE "This should never print"
```

In the following example, the second and third labels identify procedure blocks (a label specified with parameter parentheses). Execution stops when encountering a procedure block label:

ObjectScript

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",! }
  ELSE {DO label1
           WRITE "Finished Routine1",! }
  QUIT
label0
  WRITE "In Routine0",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine0",!
label1()
  WRITE "In Routine1",!
  FOR i=1:1:5 {
      WRITE "x = ",x,!
      SET x = x+1 }
  WRITE "At the end of Routine1",!
label2()
  WRITE "This should never print"
```