Using ObjectScript

Version 2024.1
2024-07-02

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

Using ObjectScript
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

For Support questions about any InterSystems products, contact:

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

Table of Contents

1 Introducing ObjectScript 1
- 1.1 Features 1
- 1.2 Language Overview 2
- 1.3 Introduction to Language Elements 2
  - 1.3.1 Statements and Commands 3
  - 1.3.2 Functions 3
  - 1.3.3 Expressions 4
  - 1.3.4 Variables 4
  - 1.3.5 Operators 5
- 1.4 See Also 5

2 Syntax Rules 7
- 2.1 Left-to-Right Precedence 7
- 2.2 Case Sensitivity 7
  - 2.2.1 Identifiers 8
  - 2.2.2 Keyword Names 8
  - 2.2.3 Class Names 8
  - 2.2.4 Namespace Names 8
- 2.3 Unicode 8
  - 2.3.1 Letters in Unicode 9
  - 2.3.2 List Compression 9
- 2.4 Whitespace 9
- 2.5 Comments 10
  - 2.5.1 Comments in INT Code for Routines and Methods 10
  - 2.5.2 Comments in MAC Code for Routines and Methods 11
  - 2.5.3 Comments in Class Definitions Outside of Method Code 11
- 2.6 String Literals 11
- 2.7 Numeric Literals 13
- 2.8 Identifiers 14
  - 2.8.1 Punctuation Characters within Identifiers 14
- 2.9 Labels 14
  - 2.9.1 Using Labels 15
  - 2.9.2 Ending a Labelled Section of Code 15
- 2.10 Namespaces 17
  - 2.10.1 Extended References 17
- 2.11 Reserved Words 18

3 Data Types and Values 19
- 3.1 Strings 19
  - 3.1.1 Null String / $CHAR(0) 19
  - 3.1.2 Escaping Quotation Marks 20
  - 3.1.3 Concatenating Strings 20
  - 3.1.4 String Comparisons 20
  - 3.1.5 Bit Strings 21
- 3.2 Numbers 22
  - 3.2.1 Fundamentals of Numbers 22
  - 3.2.2 Canonical Form of Numbers 23
  - 3.2.3 Strings as Numbers 24

3.2.4 Concatenating Numbers 26
3.2.5 Floating Point Numbers 26
3.2.6 Scientific Notation 28
3.2.7 Extremely Large Numbers 28
3.3 Objects 30
3.4 Persistent Multidimensional Arrays (Globals) 31
3.5 Undefined Values 31
3.6 Boolean Values 32
3.7 Dates 32

4 Variables 35
4.1 Categories of Variables 35
4.2 Local Variables 35
4.2.1 Naming Conventions 36
4.2.2 Scope of Local Variables 36
4.3 Globals 37
4.4 Process-Private Globals 38
4.4.1 Naming Conventions 38
4.4.2 Listing Process-Private Globals 39
4.5 Rules About Subscripts 40
4.6 Variable Typing and Conversion 40
4.7 #dim (Optional) 41
4.8 Global Variables and Journaling 41
4.9 Special Variables 42

5 Operators and Expressions 43
5.1 Introduction to Operators and Expressions 43
5.1.1 Assignment 43
5.2 Operator Precedence 44
5.2.1 Unary Negative Operators 44
5.2.2 Parentheses and Precedence 44
5.2.3 Functions and Precedence 45
5.3 String-to-Number Conversion 46
5.3.1 Numeric Strings 46
5.3.2 Non-Numeric Strings 47
5.4 Expressions 47
5.4.1 Logical Expressions 49
5.5 Arithmetic Operators 50
5.6 Numeric Relational Operators 52
5.7 Logical Comparison Operators 53
5.7.1 Precedence and Logical Operators 53
5.7.2 Logical Operators 54
5.8 String Concatenate Operator (_) 55
5.9 String Relational Operators 55
5.10 Pattern Match Operator (?) 56
5.11 Indirection Operator (@) 57

6 Commands 59
6.1 Command Keywords 59
6.2 Command Arguments 60
6.2.1 Multiple Arguments 60
6.2.2 Arguments with Parameters and Postconditionals 61

6.2.3 Argumentless Commands 61
6.3 Command Postconditional Expressions 62
6.3.1 Postconditional Syntax 63
6.3.2 Evaluation of Postconditionals 63
6.4 Multiple Commands on a Line 64
6.5 Variables 64
6.6 Error Processing 65
6.7 Transaction Processing 65
6.8 Locking and Concurrency Control 66
6.9 Invoking Code 66
6.9.1 DO 66
6.9.2 JOB 67
6.9.3 XECUTE 67
6.9.4 QUIT and RETURN 67
6.10 Controlling Flow 68
6.10.1 Conditional Execution 68
6.10.2 FOR 69
6.10.3 WHILE and DO WHILE 71
6.11 Controlling I/O 71
6.11.1 Display (Write) Commands 71
6.11.2 READ 75
6.11.3 OPEN, USE, and CLOSE 75

7 Callable User-defined Code Modules 77
7.1 Procedures, Routines, Subroutines, Functions, and Methods: What Are They? 78
7.1.1 Routines 78
7.1.2 Subroutines 79
7.1.3 Functions 80
7.2 Defining Procedures 81
7.2.1 Invoking Procedures 81
7.2.2 Procedure Syntax 82
7.2.3 Procedure Variables 83
7.2.4 Public and Private Procedures 85
7.3 Parameter Passing 86
7.3.1 Passing By Value 87
7.3.2 Passing By Reference 87
7.3.3 Variable Number of Parameters 88
7.4 Procedure Code 90
7.5 Indirection, XECUTE Commands, and JOB Commands within Procedures 92
7.6 Error Traps within Procedures 92
7.7 Legacy User-Defined Code 93
7.7.1 Subroutines 93
7.7.2 Functions 94

8 Using Macros and Include Files 99
8.1 Macro Basics 99
8.2 Include File Basics 100
8.3 Defining Macros 100
8.3.1 Where to Define Macros 101
8.3.2 Allowed Macro Definitions 101
8.3.3 Macro Naming Conventions 102
8.3.4 Macro Whitespace Conventions 102

8.3.5 Macro Comments and Studio Assist ... 102
8.4 Including Include Files ... 103
8.5 Where to See Expanded Macros ... 104
8.6 See Also ... 104

9 Embedded SQL ... 105
9.1 Embedded SQL ... 105
9.2 Other Forms of Queries ... 105

10 Multidimensional Arrays ... 107
10.1 What Multidimensional Arrays Are ... 107
10.1.1 Multidimensional Tree Structures ... 107
10.1.2 Sparse Multidimensional Storage ... 108
10.1.3 Kinds of Multidimensional Arrays ... 108
10.2 Manipulating Multidimensional Arrays ... 108
10.3 See Also ... 109

11 String Operations ... 111
11.1 Basic String Operations and Functions ... 111
11.1.1 Advanced Features of $EXTRACT ... 112
11.2 Delimited Strings ... 113
11.2.1 Advanced $PIECE Features ... 114
11.3 List-Structure String Operations ... 114
11.3.1 Sparse Lists and Sublists ... 116
11.4 Lists and Delimited Strings Compared ... 116
11.4.1 Advantages of Lists ... 116
11.4.2 Advantages of Delimited Strings ... 116

12 Locking and Concurrency Control ... 119
12.1 Introduction ... 119
12.2 Lock Names ... 119
12.3 The Lock Table ... 120
12.4 Locks and Arrays ... 120
12.5 Using the LOCK Command ... 121
12.5.1 Adding an Incremental Lock ... 121
12.5.2 Adding an Incremental Lock with a Timeout ... 122
12.5.3 Removing a Lock ... 122
12.5.4 Other Basic Variations of the LOCK Command ... 123
12.6 Lock Types ... 123
12.6.1 Exclusive and Shared Locks ... 124
12.6.2 Non-Escalating and Escalating Locks ... 124
12.6.3 Summary of Lock Types ... 124
12.7 Escalating Locks ... 125
12.7.1 Lock Escalation Example ... 125
12.7.2 Removing Escalating Locks ... 126
12.8 Locks, Globals, and Namespaces ... 127
12.8.1 Scenario 1: Multiple Namespaces with the Same Globals Database ... 127
12.8.2 Scenario 2: Namespace Uses a Mapped Global ... 128
12.8.3 Scenario 3: Namespace Uses a Mapped Global Subscript ... 129
12.8.4 Scenario 4: Extended Global References ... 130
12.9 Avoiding Deadlock ... 130
12.10 Practical Uses for Locks ... 131
12.10.1 Controlling Access to Application Data ... 131

12.10.2 Preventing Simultaneous Activity ... 131
12.11 Locking and Concurrency in SQL and Persistent Classes ... 132
12.12 See Also ... 132

13 Details of Lock Requests and Deadlocks ... 135
13.1 Waiting Lock Requests ... 135
13.2 Queuing of Array Node Lock Requests ... 136
13.3 ECP Local and Remote Lock Requests ... 137
13.4 Avoiding Deadlock ... 137
13.5 See Also ... 138

14 Managing the Lock Table ... 139
14.1 Available Tools for Managing the Lock Table ... 139
14.2 Viewing Locks in the Management Portal ... 139
14.3 Removing Locks in the Management Portal ... 142
14.4 ^LOCKTAB Utility ... 142
14.5 See Also ... 143

15 Transaction Processing ... 145
15.1 About Transactions in InterSystems IRIS ... 145
15.2 Managing Transactions Within Applications ... 145
15.2.1 Transaction Commands ... 146
15.2.2 Using LOCK in Transactions ... 146
15.2.3 Using $INCREMENT and $SEQUENCE in Transactions ... 147
15.2.4 Transaction Rollback within an Application ... 147
15.2.5 Examples of Transaction Processing Within Applications ... 148
15.3 Automatic Transaction Rollback ... 149
15.4 System-Wide Issues with Transaction Processing ... 149
15.4.1 Backups and Journaling with Transaction Processing ... 149
15.4.2 Asynchronous Error Notifications ... 149
15.5 Suspending All Current Transactions ... 150

16 Working with %Status Values ... 151
16.1 Basics of Working with Status Values ... 151
16.2 Examples ... 152
16.3 Variation (%objlasterror) ... 152
16.4 Multiple Errors Reported in a Status Value ... 152
16.5 Returning a %Status ... 153
16.6 %SYSTEM.Error ... 154
16.7 See Also ... 154

17 Using TRY-CATCH ... 155
17.1 Introduction ... 155
17.2 Using THROW with TRY-CATCH ... 156
17.3 Using $$$ThrowOnError and $$$ThrowStatus Macros ... 157
17.4 Using the %Exception.SystemException and %Exception.AbstractException Classes ... 158
17.5 Other Considerations with TRY-CATCH ... 158
17.5.1 QUIT within a TRY-CATCH Block ... 158
17.5.2 TRY-CATCH and the Execution Stack ... 159
17.5.3 Using TRY-CATCH with Traditional Error Processing ... 159

18 Error Logging ... 161
18.1 Logging Application Errors ... 161
18.2 Using Management Portal to View Application Error Logs ... 161

18.3 Using ^%ERN to View Application Error Logs ... 162
18.4 See Also ... 163

19 Command-Line Routine Debugging ... 165
19.1 Secure Debug Shell ... 165
19.1.1 Restricted Commands and Functions ... 166
19.2 Debugging with the ObjectScript Debugger ... 167
19.2.1 Using Breakpoints and Watchpoints ... 168
19.2.2 Establishing Breakpoints and Watchpoints ... 168
19.2.3 Disabling Breakpoints and Watchpoints ... 172
19.2.4 Delaying Execution of Breakpoints and Watchpoints ... 173
19.2.5 Deleting Breakpoints and Watchpoints ... 173
19.2.6 Single-step Breakpoint Actions ... 173
19.2.7 Tracing Execution ... 174
19.2.8 INTERRUPT Keypress and Break ... 175
19.2.9 Displaying Information About the Current Debug Environment ... 175
19.2.10 Using the Debug Device ... 177
19.2.11 ObjectScript Debugger Example ... 178
19.2.12 Understanding ObjectScript Debugger Errors ... 179
19.3 Debugging With BREAK ... 179
19.3.1 Using Argumentless BREAK to Suspend Routine Execution ... 180
19.3.2 Using Argumented BREAK to Suspend Routine Execution ... 180
19.3.3 Terminal Prompt Shows Program Stack Information ... 181
19.3.4 FOR Loop and WHILE Loop ... 182
19.3.5 Resuming Execution after a BREAK or an Error ... 182
19.3.6 The NEW Command at the Terminal Prompt ... 184
19.3.7 The QUIT Command at the Terminal Prompt ... 184
19.3.8 InterSystems IRIS Error Messages ... 184
19.4 Using %STACK to Display the Stack ... 185
19.4.1 Running %STACK ... 185
19.4.2 Displaying the Process Execution Stack ... 185
19.4.3 Understanding the Stack Display ... 186
19.5 Other Debugging Tools ... 191
19.5.1 Displaying References to an Object with $SYSTEM.OBJ.ShowReferences ... 191
19.5.2 Error Trap Utilities ... 191

Appendix A: (Legacy) Using ^%ETN for Error Logging ... 193

Appendix B: (Legacy) Traditional Error Processing ... 195
B.1 How Traditional Error Processing Works ... 195
B.1.1 Internal Error-Trapping Behavior ... 195
B.1.2 Current Context Level ... 196
B.1.3 Error Codes ... 197
B.2 Handling Errors with $ZTRAP ... 199
B.2.1 Setting $ZTRAP in a Procedure ... 199
B.2.2 Setting $ZTRAP in a Routine ... 199
B.2.3 Writing $ZTRAP Code ... 200
B.2.4 Using $ZTRAP ... 200
B.2.5 Unstacking NEW Commands With Error Traps ... 200
B.2.6 $ZTRAP Flow of Control Options ... 201
B.3 Handling Errors with $ETRAP ... 202
B.3.1 $ETRAP Error Handlers ... 203

B.3.2 Context-specific $ETRAP Error Handlers 203
B.3.3 $ETRAP Flow of Control Options 204
B.4 Handling Errors in an Error Handler 205
B.4.1 Errors in a $ZTRAP Error Handler 205
B.4.2 Errors in a $ETRAP Error Handler 206
B.4.3 Error Information in the $ZERROR and $ECODE Special Variables 206
B.5 Forcing an Error 206
B.5.1 Setting $ECODE 206
B.5.2 Creating Application-Specific Errors 207
B.6 Processing Errors at the Terminal Prompt 207
B.6.1 Understanding Error Message Formats 207
B.6.2 Understanding the Terminal Prompt 208
B.6.3 Recovering from the Error 208

List of Figures

Figure II–1: Frames on a Call Stack 196
Figure II–2: $ZTRAP Error Handlers 202
Figure II–3: $ETRAP Error Handlers 204

List of Tables

Table 3–1: Date Formats 33
Table 4–1: ObjectScript Type Conversion Rules 41
Table 6–1: Display Formatting 72
Table 6–2: How Values are Displayed 73
Table 15–1: Transaction Commands 146
Table 19–1: Stack Error Codes at the Terminal Prompt 181
Table 19–2: %STACK Utility Information 187
Table 19–3: Frame Types and Values Available 187

1

Introducing 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 ObjectScript routines: individual programs contained and executed within InterSystems IRIS.
- Interactively from the command line of 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 Language Overview

ObjectScript does not define any reserved words: you are free to use any word as an identifier (such as a variable name). In order to accomplish this, ObjectScript uses a set of built-in commands as well as special characters (such as the "$" prefix for function names) in order to distinguish identifiers from other language elements.

For example, to assign a value to a variable, you can use the **SET** command:

ObjectScript

```
SET x = 100
WRITE x
```

In ObjectScript it is possible (though not recommended) to use any valid name as an identifier name, as shown in the following program, which is functionally identical to the previous example:

ObjectScript

```
SET SET = 100
WRITE SET
```

Some components of ObjectScript, such as command names and function names, are not case-sensitive. Other components of ObjectScript, such as variable names, labels, class names and method names are case-sensitive. For details, see Case Sensitivity.

You can insert or omit whitespace almost anywhere in ObjectScript. However, two uses of whitespace are significant:

1. A command and its arguments must be separated by at least one space.
2. Each command line must be indented by at least one space. A command cannot start or continue on the first character position of a line.

Comments must also be indented. Some other syntaxes, such as macro preprocessor statements, can begin on the first character position of a line. For details, see Whitespace.

ObjectScript does not use a command terminator character or a line terminator character.

1.3 Introduction to Language Elements

ObjectScript syntax, in its simplest form, involves invoking commands on expressions, such as:

ObjectScript

```
WRITE x
```

which invokes the **WRITE** command on the variable x (this displays the value of x). In the example above, x is an *expression*; an ObjectScript expression is one or more "tokens" that can be evaluated to yield a value. Each token can be a literal, a variable, the result of the action of one or more operators (such as the total from adding two numbers), the return value that results from evaluating a function, some combination of these, and so on. The valid syntax for a statement involves its commands, functions, expressions, and operators.

1.3.1 Statements and Commands

An ObjectScript program consists of a number of statements. Each statement defines a specific action for a program to undertake. Each statement consists of a *command* and its *arguments*.

Consider the following ObjectScript statement:

ObjectScript

```
SET x="World"
WRITE "Hello",!,x
```

WRITE is a command. It does exactly what its name implies: it writes whatever you specify as its *argument(s)* to the current principal output device. In this case, **WRITE** writes three arguments: the literal string "Hello"; the "!" character, which is a symbolic operator specific to the **WRITE** command that issues a line feed/carriage return; and the local variable *x*, which is replaced during execution by its current value. Arguments are separated by commas; you may also add whitespace between arguments (with some restrictions). For information on whitespace, see Syntax.

Most ObjectScript commands (and many functions and special variables) have a long form and a short (abbreviated) form (typically one or two characters). For example, the following program is identical to the previous one, but uses the abbreviated command names:

ObjectScript

```
S x="World"
W "Hello",!,x
```

Older code commonly used the short forms (see Abbreviations Used in ObjectScript). For clarity, it is best to use the long forms.

For more information on commands, see Commands or the individual reference page within the ObjectScript Reference.

1.3.2 Functions

A *function* is code that performs an operation (for example, converting a string to its equivalent ASCII code values) and returns a value. A function is invoked within a command line. This invocation supplies parameter values to the function, which uses these parameter values to perform some operation. The function then returns a single value (the result) to the invoking command. You can use a function any place you can use an *expression*.

InterSystems IRIS provides a large number of system-supplied functions (sometimes known as "intrinsic" functions), which you cannot modify. These functions are identifiable, as they always begin with a single dollar sign ("$") and enclose their parameters within parentheses; even when no parameters are specified, the enclosing parentheses are mandatory. (Special variable names also begin with a single dollar sign, but they do not have parentheses.)

Many system-supplied function names have abbreviations. In the text of this manual, the full function names are used. The abbreviation is shown on the function's reference page and a complete list is provided in Abbreviations Used in ObjectScript.

A function always returns a value. Commonly, this return value is supplied to a command, such as `SET namelen=$LENGTH("Fred Flintstone")` or `WRITE $LENGTH("Fred Flintstone")`, or to another function, such as `WRITE $LENGTH($PIECE("Flintstone^Fred","^",1))`. Failing to provide a recipient for the return value usually results in a <SYNTAX> error. However, in a few functions, providing a recipient for the return value is not required. An operation performed by executing the function or the setting of one of the function's parameters is the relevant operation. In these cases, you can invoke a function without receiving its return value by using the **DO** or **JOB** command. For example, `DO $CLASSMETHOD(clname,clmethodname,singlearg)`.

A function can have no parameters, a single parameter, or multiple parameters. Function parameters are positional and separated by commas. Many parameters are optional. If you omit a parameter, InterSystems IRIS uses that parameter's

default. Because parameters are positional, you commonly cannot omit a parameter within a list of specified parameters. In some cases (such as **$LISTTOSTRING**) you can omit a parameter within a parameter list and supply a placeholder comma. You do not have to supply placeholder commas for optional parameters to the right of the last specified parameter.

For most functions, you cannot specify multiple instances of the same parameter. The exceptions are **$CASE**, **$CHAR**, and **$SELECT**.

Commonly, a parameter can be specified as a literal, a variable, or the return value of another function. In a few cases, a parameter must be supplied as a literal. In most cases, a variable must be defined before it can be specified as a function parameter, or an <UNDEFINED> error is generated. In a few cases (such as **$DATA**) the parameter variable does not have to be defined.

Commonly, function parameters are input parameters that supply a value to the function. In a few cases, a function both returns a value and sets an output parameter. For example, **$LISTDATA** returns a boolean value indicating whether there is a list element at the specified position; it also sets its third parameter (if included in the parameter list) equal to the value of that list element.

All functions can be specified on the right side of a **SET** command (for example, `SET x=$LENGTH(y)`). A few functions can also be specified on the left side of a **SET** command (for example, `SET $LIST(list,position,end)=x`). Functions that can be specified on the left side of a **SET** are identified as such in their reference page syntax block.

System-supplied functions are provided as part of InterSystems IRIS. The ObjectScript Language Reference describes each of the system-supplied functions. A function provided in a class is known as a method. Methods provided in InterSystems IRIS are described in the *InterSystems Class Reference*.

In addition to its system-supplied functions, ObjectScript also supports user-defined functions (sometimes known as "extrinsic" functions). For information on defining and calling user-defined functions, refer to User-Defined Code.

1.3.3 Expressions

An expression is any set of tokens that can be evaluated to yield a single value. For example, the literal string, "hello", is an expression. So is `l + 2`. Variables such as *x*, functions such as `$LENGTH`, and special variables such as `$ZVERSION` also evaluate to an expression.

Within a program, you use expressions as arguments for commands and functions:

ObjectScript

```
SET x = "Hello"
WRITE x,!
WRITE 1 + 2,!
WRITE $LENGTH(x),!
WRITE $ZVERSION
```

1.3.4 Variables

In ObjectScript, a variable is the name of a location in which a runtime value can be stored. Variables must be defined, for example, by using the **SET** command.

Variables in ObjectScript are untyped; that is, they do not have an assigned data type and can legally take any data value. No syntax error occurs when you assign a string value to a variable that previously held a numeric value, or vice versa. (Syntax errors do occur, however, if you attempt to use a variable inappropriately, such as if you try to set an object property when the variable does not contain an instance of an object , or when you pass a non-list value to a function that requires a list, and so on. That is, many ObjectScript functions expect specific kinds of input.)

ObjectScript supports several kinds of variables, characterized by differing scopes and features:

- Local variables — A variable that is accessible only by the InterSystems IRIS process that created it, and which is automatically deleted when the process terminates.

- Process-private globals — A variable that is accessible only by the InterSystems IRIS process and is deleted when the process ends. Process-private globals are especially useful for temporary storage of large data values.
- Globals — A persistent variable that is stored within the InterSystems IRIS database. A global is accessible from any process, and persists after the process that created it terminates.
- Array variables — A variable with one or more subscripts. All user-defined variables can be used as arrays, including local variables, process-private globals, globals, and object properties.
- Special variables (also known as system variables) — One of a special set of built-in variables that contain a value for a particular aspect of the InterSystems IRIS operating environment. All special variables always have values. Some special variables can be set by the user, others can only be set by InterSystems IRIS. Special variables are not array variables.
- Object properties — A value associated with, and stored within, a specific instance of an object.

ObjectScript supports various operations on or among variables. Variables are described in more detail in Variables.

1.3.5 Operators

ObjectScript defines a number of built-in operators. These include arithmetic operators, logical operators, and pattern match operators. For details, see Operators.

1.4 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

Syntax Rules

This topic describes the basic rules of ObjectScript syntax.

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. Generally speaking, the user-definable parts of ObjectScript are case-sensitive while keywords are not:

- *Case-sensitive*: variable names (local, global, and process-private global) and variable subscripts, class names, method names, property names, the i% preface for an instance variable for a property, routine names, macro names, macro include file (.inc file) names, label names, lock names, passwords, embedded code directive marker strings, Embedded SQL host variable names.
- *Not case-sensitive*: command names, function names, special variable names, namespace names (see below), user names and role names, preprocessor directives (such as #include), letter codes (for LOCK, OPEN, or USE), keyword codes (for $STACK), pattern match codes, and embedded code directives (&html, &js, &sql). Custom language elements that you add by customizing the %ZLANG routine are not case-sensitive; when you create them you must use uppercase, when you refer to them you can use any case. Because indexing for text analytics normalizes text by converting it to lowercase, most NLP values, including domain names, 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. Uppercase “E” is always a valid exponent symbol; lowercase “e” can be configured as valid or invalid for the current process using the **ScientificNotation()** method of the %SYSTEM.Process, or system-wide using the *ScientificNotation* property of the Config.Miscellaneous class.

2.2.1 Identifiers

User-defined identifiers (variable, routine, and label names) are case-sensitive. *String*, *string*, and *STRING* all refer to different variables. Global variable names are also case-sensitive, whether user-defined or system-supplied.

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

2.2.2 Keyword Names

Command, function, and system variable keywords (and their abbreviations) are not case-sensitive. You can use **Write**, **write**, **WRITE**, **W**, or **w**; all refer to the same command.

2.2.3 Class Names

All identifiers related to classes (class names, property names, method names, etc.) are case-sensitive. For purposes of uniqueness, however, such names are considered to be not case-sensitive; that is, two class names cannot differ by case alone.

2.2.4 Namespace Names

Namespace names are not case-sensitive, meaning that you can input a namespace name in any combination of uppercase and lowercase letters. Note however, that InterSystems IRIS® data platform always stores namespace names in uppercase. Therefore, InterSystems IRIS may return a namespace name to you in uppercase rather than in the case which you specified. For further details on namespace naming conventions, see Namespaces.

2.3 Unicode

InterSystems IRIS supports the Unicode international character set. Unicode characters are 16-bit characters, also known as wide characters. The $ZVERSION special variable (`Build nnnU`) and the **$SYSTEM.Version.IsUnicode()** method show that the InterSystems IRIS installation supports Unicode.

For most purposes, InterSystems IRIS only supports the Unicode Basic Multilingual Plane (hex 0000 through FFFF) which contains the most commonly-used international characters. Internally, InterSystems IRIS uses the UCS-2 encoding, which for the Basic Multilingual Plane, is the same as UTF-16. You can work with characters that are not in the Unicode Basic Multilingual Plane by using $WCHAR, $WISWIDE, and related functions.

InterSystems IRIS encodes Unicode strings into memory by allocating 16 bits (two bytes) per character, as is standard with UTF-16 encodings. However, when saving a Unicode string to a global, if all characters have numerical values of 255 or lower, InterSystems IRIS stores the string using 8 bits (one byte) per character. If the string contains characters with numerical values greater than 255, InterSystems IRIS applies a compression algorithm to reduce the amount of space the string takes up in storage.

For conversion between Unicode and UTF-8, and conversions to other character encodings, refer to the $ZCONVERT function. You can use ZZDUMP to display the hexadecimal encoding for a string of characters. You can use $CHAR to specify a character (or string of characters) by its decimal (base 10) encoding. You can use $ZHEX to convert a hexadecimal number to a decimal number, or a decimal number to a hexadecimal number.

2.3.1 Letters in Unicode

On InterSystems IRIS, some names can contain Unicode letter characters, while other names cannot contain Unicode letters. Unicode letters are defined as alphabetic characters with decimal character code values higher than 255. For example, the Greek lowercase lambda is $CHAR(955), a Unicode letter.

Unicode letter characters are permitted throughout InterSystems IRIS, with the following exceptions:

- Variable names: local variable names can contain Unicode letters. However, global variable names and process-private global names cannot contain Unicode letters. Subscripts for variables of all types can be specified with Unicode characters.
- Administrator user names and passwords used for database encryption cannot contain Unicode characters.

The locale identifier is not taken into account when dealing with Unicode characters. That is, if a identifier consisting of Unicode characters is valid in one locale, the identifier is valid in any locale. Note that the above exceptions still apply.

Note: The Japanese locale does not support accented Latin letter characters in InterSystems IRIS names. Japanese names may contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), and the Greek capital letter characters (913–929 and 931–937).

2.3.2 List Compression

ListFormat 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**.

2.4 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:
  - 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.

2.5 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

2.5.1 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"
```

2.5.2 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.

2.5.3 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.

2.6 String Literals

A string literal is a set of zero or more characters delimited by quotation marks (in contrast, numeric literals do not need a surrounding pair of delimiters). ObjectScript string literals are delimited with double quotation marks (for example, `"myliteral"`); InterSystems SQL string literals are delimited with single quotation marks (for example, `'myliteral'`). These quotation mark delimiters are not counted in the length of the string.

A string literal can contain any characters, including whitespace and control characters. There is a maximum permitted length (see String Length Limit). If a string contains only characters with codes from 0 to 255 (also known as Latin-1 or ASCII Extended characters), then each characters takes up 8 bits (one byte). If a string contains at least one character with a code greater than 255 (also known as Unicode or wide characters), then each character takes up 16 bits (two bytes). To view the bytes used to store string characters, you can use the **ZZDUMP** command, as shown in the next example.

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
```

Not all string characters are typeable. You can specify non-typeable characters using the **$CHAR** function, as shown in the following Unicode example:

ObjectScript

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

Not all string characters are displayable. They can be non-printing characters or control characters. The **WRITE** command represents non-printing characters as a box symbol. The **WRITE** command causes control characters to execute. In the following example, a string contains printable characters alternating with the Null ($CHAR(0)), Tab ($CHAR(9)) and Carriage Return ($CHAR(13)) characters:

ObjectScript

```
  SET a="a"_$CHAR(0)_"b"_$CHAR(9)_"c"_$CHAR(13)_"d"
  WRITE !,"the length of string a is ",$LENGTH(a)
  ZZDUMP a
  WRITE !,a
```

Note that the **WRITE** command executes some control characters from the Terminal command line which **WRITE** executing in a program displays as non-printing characters. For example, the Bell ($CHAR(7)) and Vertical Tab ($CHAR(11)) characters.

To include the quotation mark character (") within a string, double the character, as shown in the following example:

ObjectScript

```
  SET x="This quote"
  SET y="This "" quote"
  WRITE x,!," string length=",$LENGTH(x)
  ZZDUMP x
  WRITE !!,y,!," string length=",$LENGTH(y)
  ZZDUMP y
```

A string that contains no value is known as a null string. It is represented by two quotation mark characters (""). A null string is considered to be a defined value. It has a length of 0. Note that the null string is *not* the same as a string consisting of the 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
```

For further details on strings, see Strings.

2.7 Numeric Literals

Numeric literals are values that ObjectScript evaluates as numbers. In contrast to string literals, they do not require a surrounding pair of delimiters. InterSystems IRIS converts numeric literals to canonical form (their simplest numeric form):

ObjectScript

```
SET x = ++0007.00
WRITE "length:     ",$LENGTH(x),!
WRITE "value:      ",x,!
WRITE "equality:   ",x = 7,!
WRITE "arithmetic: ",x + 1
```

You *can* also represent a number as a string literal delimited with quotation marks; a numeric string literal is not converted to canonical form, but can be used as a number in arithmetic operations:

ObjectScript

```
SET y = "++0007.00"
WRITE "length:     ",$LENGTH(y),!
WRITE "value:      ",y,!
WRITE "equality:   ",y = 7,!
WRITE "arithmetic: ",y + 1
```

For further details refer to Strings as Numbers.

ObjectScript treats as a number any value that contains the following (and no other characters):

Value	Quantity
The digits 0 through 9.	Any quantity, but at least one.
Sign operators: Unary Minus (-) and Unary Plus (+).	Any quantity, but must precede all other characters.
The decimal_separator character (by default this is the period or decimal point character; in European locales this is the comma character).	At most one.
The Letter E (used in scientific notation).	At most one. Must appear between two numbers.

For further details on the use and interpretation of these characters, refer to Fundamentals of Numbers.

ObjectScript can work with the following types of numbers:

- Integers (whole numbers such as 100, 0, or -7).

- Fractional numbers: decimal numbers (real numbers such as 3.767) and decimal fractions (real numbers such as .0442). 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). For further details, refer to the $DOUBLE function.
- Scientific notation: numbers placed in exponential notation (such as 2.8E2).

2.8 Identifiers

An *identifier* is the name of a variable, a routine, or a 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.

The naming conventions for user-defined commands, functions, and special variables are more restrictive (only letters permitted) than identifier naming conventions. See Extending Languages with ^%ZLANG Routines.

For naming conventions for local variables, process-private globals, and globals, see Variables.

2.8.1 Punctuation Characters within Identifiers

Certain identifiers can contain one or more punctuation characters. These include:

- The first character of an identifier can be a percent (%) character. InterSystems IRIS names beginning with a % character (except those beginning with %Z or %z) are reserved as system elements. For further details, see Rules and Guidelines for Identifiers.
- A global or process-private global name (but not a local variable name) may include one or more period (.) characters. A routine name may include one or more period (.) characters. A period cannot be the first or last character of an identifier.

Note that globals and process-private globals are identified by a caret (^) prefix of one or more characters, such as the following:

```
  Globals:                                  Process-Private Globals:
^globname                                   ^||ppgname
^|""|globname                               ^|"^"|ppgname
^|"mynspace"|globname                    ^|"^",""|ppgname
^["mynspace"]globname                 ^["^"]ppgname
```

These prefix characters are not part of the variable name; they identify the type of storage and (in the case of globals) the namespace used for this storage. The actual name begins after the final vertical bar or closing square bracket.

2.9 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
86
agent86
86agent
%control
```

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

You can use the ZINSERT command to insert a label name into source code.

2.9.1 Using Labels

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

The DO and GOTO commands can specify their target location as a label. The $ZTRAP special variable can specify the location of its error handler as a label. The JOB command can specify the routine to be executed as a label.

Labels are also used by the PRINT, ZPRINT, ZZPRINT, ZINSERT, ZREMOVE, and ZBREAK commands and the $TEXT function to identify source code lines.

However, you cannot specify a label on the same line of code as a CATCH command, or between a **TRY** block and a **CATCH** block.

2.9.2 Ending a Labelled Section of Code

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"
```

2.10 Namespaces

A namespace name may be an explicit namespace name or an implied namespace name. An explicit namespace name is not case-sensitive; regardless of the case of the letters with which it is input, it is always stored and returned in uppercase letters.

In an explicit namespace name, the first character must be a letter or a percent sign (%). The remaining characters must be letters, numbers, hyphens (–), or underscores (_). The name cannot be longer than 255 characters.

When InterSystems IRIS translates an explicit namespace name to a routine or class name (for example, when creating a cached query class/routine name), it replaces punctuation characters with lowercase letters, as follows: % = p, _ = u, – = d. An implied namespace name may contain other punctuation characters; when translating an implied namespace name, these punctuation characters are replaced by a lowercase "s". Thus the following seven punctuation characters are replaced as follows: @ = s, : = s, / = s, \ = s, [= s,] = s, ^ = s.

The following namespace names are reserved: %SYS, BIN, BROKER, and DOCUMATIC.

When using the InterSystems SQL CREATE DATABASE command, creating an SQL database creates a corresponding InterSystems IRIS namespace.

A namespace exists as a directory in your InterSystems IRIS instance. To return the full pathname of the current namespace, you can invoke the **NormalizeDirectory()** method, as shown in the following example:

ObjectScript

```
   WRITE ##class(%Library.File).NormalizeDirectory("")
```

For information on using namespaces, see Namespaces and Databases. For information on creating namespaces, see Configuring Namespaces.

2.10.1 Extended References

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. There are three types of extended references:

- Extended Global Reference: references a global variable in another namespace. The following syntactic forms are supported: `^["namespace"]global` and `^|"namespace"|global`. For further details, see Global Variables.

- 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"]`, or `routine:"namespace"`.

  In all these cases, the extended routine 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.
- Extended SSVN Reference: references a structured system variable (SSVN) in another namespace. The following syntactic forms are supported: `^$["namespace"]ssvn` and `^$|"namespace"|ssvn`. For further details, refer to the ^$GLOBAL, ^$LOCK, and ^$ROUTINE structured system variables.

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

2.11 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.

3

Data Types and Values

ObjectScript is a typeless language — you do not have to declare the types of variables. Any variable can have a string, numeric, or object value. That being said, there is important information to know when using different kinds of data in ObjectScript.

3.1 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
```

Topics about strings include:

- Null String / $CHAR(0)
- Escaping Quotation Marks
- Concatenating Strings
- String Comparisons
- Bit Strings

Also see String Length Limit.

3.1.1 Null String / $CHAR(0)

- **SET mystr=""**: sets a null or empty string. The string is defined, is of zero length, and contains no data:

  ObjectScript

  ```
  SET mystr=""
  WRITE "defined:",$DATA(mystr),!
  WRITE "length: ",$LENGTH(mystr),!
  ZZDUMP mystr
  ```

- **SET mystr=$CHAR(0)**: sets a string to the null character. The string is defined, is of length 1, and contains a single character with the hexadecimal value of 00:

ObjectScript

```
SET mystr=$CHAR(0)
WRITE "defined:",$DATA(mystr),!
WRITE "length: ",$LENGTH(mystr),!
ZZDUMP mystr
```

Note that these two values are not the same. However, a bitstring treats these values as identical.

Note that InterSystems SQL has its own interpretation of these values; see NULL and the Empty String.

3.1.2 Escaping Quotation Marks

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
```

There are no other escape character sequences within ObjectScript string literals.

Note that literal quotation marks are specified using other escape sequences in other InterSystems software. Refer to the $ZCONVERT function for a table of these escape sequences.

3.1.3 Concatenating Strings

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
```

By using the concatenate operator you can include non-printing characters in a string. The following string includes the linefeed ($CHAR(10)) character:

ObjectScript

```
SET lf = $CHAR(10)
SET string = "This"_lf_"is"_lf_"a string"
WRITE string
```

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.

InterSystems IRIS encoded strings — bit strings, List structure strings, and JSON strings — have limitations on their use of the concatenate operator. For further details, see Concatenate Encoded Strings.

Some additional considerations apply when concatenating numbers. For further details, see "Concatenating Numbers".

3.1.4 String Comparisons

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
```

The <, >, <=, or >= operators cannot be used to perform a string comparison. These operators treat strings as numbers and always perform a numeric comparison. Any non-numeric string is assigned a numeric value of 0 when compared using these operators.

3.1.4.1 Lettercase and String Comparisons

String equality comparisons are case-sensitive. You can use the $ZCONVERT function to convert the letters in the strings to be compared to all uppercase letters or all lowercase letters. Non-letter characters are unchanged.

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.

3.1.5 Bit Strings

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)**.

3.2 Numbers

Topics related to numbers include:

- Fundamentals of Numbers
- Canonical Form of Numbers
- Strings as Numbers
- Concatenating Numbers
- Floating Point Numbers
- Scientific Notation
- Extremely Large Numbers

3.2.1 Fundamentals of Numbers

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.

3.2.2 Canonical Form of 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.

3.2.3 Strings as Numbers

The following are the general rules for handling strings as numbers. For further details, see String-to-Number Conversion. For special processing, see Extremely Large Numeric Strings.

- For all numeric operations, a string containing a number in canonical form is functionally identical to the corresponding number. For example, `"3" = 3`, `"-2.5" = -2.5`. (Note that -0 is not a canonical number.)
- For arithmetic operations, a string containing only numeric characters in non-canonical form is functionally identical to the corresponding number. For example, `"003" + 3 = 6`, `"++-2.5000" + -2.5 = -5`.
- For greater-than/less-than operations, a string containing only numeric characters in non-canonical form is functionally identical to the corresponding number. For example, the following statements are true: `"003" > 2`, `"++-2.5000" >= -2.5`.
- For equality operations (=, '=), a string containing only numeric characters in non-canonical form is treated as a string, not a number. For example, the following statements are true: `"003" = "003"`, `"003" '= 3`, `"+003" '= "003"`.

Some further guidelines concerning parsing strings as numbers:

- A mixed numeric string is a string that begins with numeric characters, followed by one or more non-numeric characters. For example “7 dwarves”. InterSystems IRIS numeric and boolean operations (other than equality operations) commonly parse a mixed numeric string as a number until they encounter a non-numeric character. At that point the rest of the string is ignored. The following example shows arithmetic operations on mixed numeric strings:

  ObjectScript

  ```
  WRITE "7dwarves" + 2,!   // returns 9
  WRITE "+24/7" + 2,!      // returns 26
  WRITE "7,000" + 2,!      // returns 9
  WRITE "7.0.99" + 2,!     // returns 9
  WRITE "7.5.99" + 2,!     // returns 9.5
  ```

- A non-numeric string is any string in which a non-numeric character is encountered before encountering a numeric character. Note that a blank space is considered a non-numeric character. InterSystems IRIS numeric and boolean operations (other than equality operations) commonly parse this string as having a numeric value of 0 (zero). The following example shows arithmetic operations on non-numeric strings:

  ObjectScript

  ```
  WRITE "dwarves 7" + 2,!   // returns 2
  WRITE "+ 24/7" + 2,!      // returns 2
  WRITE "$7000" + 2,!       // returns 2
  ```

- You can prefix a string with a plus sign to force its evaluation as a number for equality operations. A numeric string is parsed as a number in canonical form; a non-numeric string is parsed as 0. (A minus sign prefix also forces evaluation of a string as a number for equality operations; the minus sign, of course, inverts the sign for a non-zero value.) The following example shows the plus sign forcing numeric evaluation for equality operations:

  ObjectScript

  ```
  WRITE +"7" = 7,!            // returns 1 (TRUE)
  WRITE +"+007" = 7,!         // returns 1 (TRUE)
  WRITE +"7 dwarves" = 7,!    // returns 1 (TRUE)
  WRITE +"dwarves" = 0,!      // returns 1 (TRUE)
  WRITE +"" = 0,!             // returns 1 (TRUE)
  ```

Numeric string handling exceptions for individual commands and functions are common, as noted in the ObjectScript Reference.

3.2.3.1 Extremely Large Numeric 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.

3.2.4 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.

3.2.5 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(12345678901234567818181818181)
  12345678901234567820000000
  ```

- 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.

3.2.6 Scientific Notation

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

```
[-]mantissaE[-]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.

3.2.7 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 1.797693134862315708E308 (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 when using InterSystems IRIS decimal numbers or IEEE double numbers, see Numeric Computing in InterSystems Applications.

3.3 Objects

An object value refers to an instance of an in-memory object. You can assign an object reference (OREF) to any local variable:

ObjectScript

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

To refer to the methods and properties of an object instance, use dot syntax:

ObjectScript

```
SET myperson.Name = "El Vez"
```

To determine if a variable contains an object, use the $ISOBJECT function:

ObjectScript

```
SET str = "A string"
SET myperson = ##class(Sample.Person).%New()

IF $ISOBJECT(myperson) {
    WRITE "myperson is an object.",!
} ELSE {
    WRITE "myperson is not an object."
}

IF $ISOBJECT(str) {
     WRITE "str is an object."
} ELSE {
    WRITE "str is not an object."
}
```

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

Assigning an object value to a variable (or object property) has the side effect of incrementing the object's internal reference count, as shown in the following example:

ObjectScript

```
SET x = ##class(Sample.Person).%New()
WRITE x,!
SET y = ##class(Sample.Person).%New()
WRITE y,!
SET z = ##class(Sample.Person).%New()
WRITE z,!
```

When the number of references to an object reaches 0, the system automatically destroys the object (invoke its %OnClose() callback method and remove it from memory).

3.4 Persistent Multidimensional Arrays (Globals)

A global is a sparse, multidimensional database array. A global is not different from any other type of array, with the exception that the global variable name starts with a caret (^). Data can be stored in a global with any number of subscripts; subscripts in InterSystems IRIS are typeless.

The following is an example of using a global. Once you set the global ^*x*, you can examine its value:

ObjectScript

```
SET ^x = 10
WRITE "The value of ^x is: ", ^x,!
SET ^x(2,3,5) = 17
WRITE "The value of ^x(2,3,5) is: ", ^x(2,3,5)
```

For more information on globals, see Multidimensional Arrays and Using Globals.

3.5 Undefined Values

ObjectScript variables do not need to be explicitly declared or defined. As soon as you assign a value to a variable, the variable is defined. Until this first assignment, all references to this variable are undefined. You can use the $DATA function to determine if a variable is defined or undefined.

$DATA takes one or two arguments. With one argument, it simply tests if a variable has a value:

ObjectScript

```
WRITE "Does ""MyVar"" exist?",!
IF $DATA(MyVar) {
        WRITE "It sure does!"
} ELSE {
       WRITE "It sure doesn't!"
}

SET MyVar = 10
WRITE !,!,"How about now?",!
IF $DATA(MyVar) {
       WRITE "It sure does!"
} ELSE {
       WRITE "It sure doesn't!"
}
```

$DATA returns a boolean that is True (1) if the variable has a value (that is, contains data) and that is False (0) if the variable has no value (that is, contains no data). With two arguments, it performs the test and sets the second argument's variable equal to the tested variable's value:

ObjectScript

```
IF $DATA(Var1,Var2) {
   WRITE "Var1 has a value of ",Var2,".",!
} ELSE {
    WRITE "Var1 is undefined.",!
}

SET Var1 = 3
IF $DATA(Var1,Var2) {
   WRITE "Var1 has a value of ",Var2,".",!
} ELSE {
    WRITE "Var1 is undefined.",!
}
```

3.6 Boolean Values

In certain cases, such as when used with logical commands or operators, a value may be interpreted as a boolean (true or false) value. In such cases, an expression is interpreted as 1 (true) if evaluates to a nonzero numeric value or 0 (false) if it evaluates to a zero numeric value. A numeric string evaluates to its numeric value; a non-numeric string evaluates to 0 (false).

For example, the following values are interpreted as true:

ObjectScript

```
IF 1 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF 8.5 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF "1 banana" { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF 1+1 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF -7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
IF +"007"=7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
```

The following values are interpreted as false:

ObjectScript

```
IF 0 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF 3-3 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "one banana" { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "" { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF -0 { WRITE "evaluates as true",! }
    ELSE { WRITE "evaluates as false",! }
IF "007"=7 { WRITE "evaluates as true",! }
     ELSE { WRITE "evaluates as false",! }
```

For further details on the evaluation of a string as a number, see String-to-Number Conversion.

3.7 Dates

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 3–1: 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,!
```

4

Variables

A variable is the name of a location in which a value can be stored. Within ObjectScript, a variable does not have data type associated with it and you do not have to declare it.

Commonly, you use the SET command to define a variable by assigning it a value. You can assign a null string ("") value to a variable. Most commands and functions require a variable to be defined before it is referenced. If the variable is undefined, by default referencing it generates an <UNDEFINED> error. You can change InterSystems IRIS® data platform behavior to not generate an <UNDEFINED> error when referencing an undefined variable by setting the **%SYSTEM.Process.Undefined()** method.

You can use an undefined variable in some operations, such as the READ command, the $INCREMENT function, the $BIT function, and the two-argument form of the $GET function. These operations assign a value to the variable. The $DATA function can take an undefined or defined variable and returns its status.

Unlike many computer languages, ObjectScript does not require variables to be declared. A variable is created when it is assigned a value. ObjectScript is a “typeless” language; a variable can receive data of any type. See Variable Declaration and Variable Typing and Conversion.

4.1 Categories of Variables

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

- Local variables
- Global variables or *globals*
- Process-private global variables or PPGs
- i%property instance variables (discussed on another page)
- Special variables or *system variables*

4.2 Local Variables

A local variable is a variable that is available in memory within the current InterSystems IRIS process. It is accessible only to the process that created it. It is accessible from all namespaces; that is, if the process changes namespaces, the variable is still available. When a process ends, all of its local variables are deleted.

4.2.1 Naming Conventions

Rules and Guidelines for Identifiers provides all the details, but briefly:

- For a local variable name, the first character must be either a letter or the percent (%) character.

  Variable names starting with the % character are known as "percent variables" and have different scoping rules. In your code, for a local percent variable, start the name with %Z or %z; other names are reserved for system use.

- All variable names are case-sensitive, and this includes local variable names. For example: `MYVAR`, `MyVar`, and `myvar` are three different local variables.
- Local variable names must be unique for the current process. Other processes may have local variables with the same name.
- A process-private global or a global may have the same (apparent) name as a local variable. For example: `myvar`, `^||myvar`, and `^myvar` are three different variables.
- Local variables can take subscripts. See Rules About Subscripts.

A local variable name that does not follow the rules generates a <SYNTAX> error. There is one exception: if an attempted variable name begins with an underscore character followed by a letter, the system generates a <_CALLBACK SYNTAX> error. For example, `SET x=_a`.

4.2.2 Scope of Local Variables

In ObjectScript code, all local variables are public, and can thus be accessed by any operation executed by that process in the current context. Access to a local variable value is restricted as follows:

- The NEW command creates a new local variable context. An argumentless **NEW** creates a new context in which none on the existing local variables are defined. **NEW var** creates a new context in which the local variable *var* is not defined. The **QUIT** command reverts to the prior local variable context.
- Within a procedure block local variables are private by default. A private local variable is only defined within that procedure block.

  Local variables within a procedure block behave as follows:

  – *Private variables*. A local variable used within a procedure block is a private variable and is only defined within that procedure block, unless it is declared a public variable or it is a % variable. By default, all object methods created with Studio use procedure blocks (the ProcedureBlock class keyword is set within the class definition) and so, by default, all variables created in methods are private variables. You cannot use the NEW command on a private variable in a procedure block.

  – *Public variables*. A procedure block can explicitly declare a list of local variables as public variables. These variables values are accessible outside the procedure block. This comma-separated list of public variables can include non-existent variables and % variables. You can use the NEW command on a public variable in a procedure block.

    A public variables list for the two local variables *var1* and *var2* is specified as follows: `MyProc(x,y) [var1,var2] PUBLIC { code body }`. (Note that the PUBLIC keyword specifies that the procedure is public; it has nothing to do with the public variables list.) Public variables are specified as a comma-separated list. Only unsubscripted local variables can be specified; specifying an unsubscripted variable in the public variables lists makes all of its subscript levels public as well. Only a simple object reference (OREF) can be specified; specifying an OREF in the public variables lists makes all of its object properties public as well. The list of public variables can include undefined variables.

  - *% Variables*. A local variable whose name starts with "%" is automatically declared a public variable. This makes it possible to define variables that are visible to all code within a process without having to explicitly list these variables as public. Only variables that begin with "%Z" or "%z" are available for application code; all other % variables are reserved for system use according to the rules described in Rules and Guidelines for Identifiers. You can use the NEW command on a % variable in a procedure block.

  These mechanisms are described in greater detail in Procedure Variables.

- The XECUTE command defines local variables as public by default. You can explicitly define a local variable as private within the **XECUTE** command. Refer to XECUTE for more on explicitly defining local variables as either private or public.

You can use the **WRITE** or **ZWRITE** command, with no arguments, to list all currently defined local variables. You can use the **KILL** command to delete local variables.

4.3 Globals

A global is a special kind of variable that is automatically stored within the InterSystems IRIS database. It is mapped to a specific namespace, and can only be accessed within that namespace, unless an extended reference is used. A global can be accessed by any process. A global persists after the termination of the process that created it. It persists until explicitly deleted.

Within ObjectScript code, you can use a global in the same way as any other variable. Syntactically, a global name is distinguished by an initial caret (^) character:

ObjectScript

```
SET mylocal = "This is a local variable"
SET ^myglobal = "This is a global stored in the current namespace"
```

For complete information on names, see Rules and Guidelines for Identifiers. Briefly:

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

  Global names starting with ^% are known as "percent variables" and are available in all namespaces. In your code, for these variables, start the name with ^%Z or ^%z; other names are reserved for system use.

- The second and subsequent characters of a global name may be letters, numbers, or the period character. A period cannot be used as the first or last character of the name.
- Unlike local variables, no global name can contain Unicode letters — letter characters above ASCII 255. Attempting to include a Unicode letter in a global name results in a <WIDE CHAR> error.
- Global names are case-sensitive.
- A global name must be unique within its namespace.
- 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 global name must be unique within its first 31 characters.
- Like other variables, globals can take subscripts. See Rules About Subscripts.
- Some global names are reserved for InterSystems use. See Global Variable Names to Avoid.

It is possible to use an extended reference to refer to a global in another namespace. The syntax uses a pair of vertical bars or square brackets immediately after the caret characters (for example: `^|"samples"|myglobal` or `^|""|myglobal`). These syntaxes should not be confused with process-private globals.

For more information on globals, refer to Using Globals.

4.4 Process-Private Globals

A 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-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.

Process-private globals are intended to be used for large data values. They can serve, in many cases, as a replacement for the use of the Mgr/Temp directory, providing automatic cleanup at process termination.

4.4.1 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. See Rules About Subscripts.

4.4.2 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")
```

4.5 Rules About Subscripts

Local variables, process-private variables, and global variables can all take subscripts. In all cases, note the following points:

- 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.
- There are limits on the length of a subscript and on the number of subscript levels. See Subscript Limits.

Also see General System Limits.

Lock name subscripts follow the same conventions as variable subscripts.

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.

4.6 Variable Typing and Conversion

Variables in ObjectScript are untyped — there are no specified data types. (This is also true of JavaScript, VBScript, and Document Data Base/JSON.) 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.

Some examples:

ObjectScript

```
// set some variables
SET a = "This is a string"
SET b = "3 little pigs"
SET int = 22
SET num = 2.2
SET obj = ##class(Sample.Person).%New()

// Display them
WRITE "Here are the variables themselves: ",!
WRITE "a: ",a,!
WRITE "b: ",b,!
WRITE "int: ",int,!
WRITE "num: ",num,!
WRITE "obj: ",obj,!,!

// Now use them as other "types"
WRITE "Here are the numeric interpretation of",!
WRITE "a, b, and obj: ",!
WRITE "+a: ",+a,!
WRITE "+b: ",+b,!
WRITE "+obj: ",+obj,!,!
```

```
WRITE "Here are concatenations of int and num:",!
WRITE "Concatenating int: ","I found " _ int _ " apples.",!
WRITE "Concatenating num: ","There are " _ num _ " pounds per kilogram.",!
```

InterSystems IRIS converts values as follows:

Table 4–1: 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.

4.7 #dim (Optional)

Unlike other languages, you do not need to declare variables in ObjectScript. You can, however, use the #dim preprocessor directive as an aid to documenting and writing code; IDEs can take advantage of this and provide code-completion assistance.

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.

4.8 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.

4.9 Special Variables

ObjectScript includes special variables (also referred to as system variables) that are used to make certain system information available to applications. All special variables are supplied with InterSystems IRIS and are named with a $ character prefix. Users cannot define additional special variables. The special variables are available in all namespaces.

The value of a special variable is set to the current state of some aspect of your operating environment. Some special variables are initially set to the null string (""); referencing a special variable should never generate an <UNDEFINED> error. The value of a special variable is specific to the current process and cannot be accessed from another process.

Users can set some special variables with the **SET** command; other special variables are not user-modifiable. Refer to the individual special variables for further details.

The following example uses the special variable $HOROLOG:

ObjectScript

```
SET starttime = $HOROLOG
HANG 5
WRITE !,$ZDATETIME(starttime)
WRITE !,$ZDATETIME($HOROLOG)
```

The special variable $HOROLOG stores the current system date and time. The **SET** command uses this special variable to set the user-defined local variable *starttime* to this value. The **HANG** command then suspends the program for 5 seconds. Finally, the two **$ZDATETIME** functions return *starttime* and the current system date and time in a user-readable format.

Other examples of special variables include:

ObjectScript

```
WRITE !,"$JOB = ",$JOB   // Current process ID
WRITE !,"$ZVERSION = ",$ZVERSION   // Version info
```

Many special variables are read-only; they cannot be set using the **SET** command. Other special variables, such as $DEVICE, are read-write, and can be set using the **SET** command.

Special variables cannot take subscripts. Special variables cannot be incremented using the **$INCREMENT** function or killed using the **KILL** command. Special variables can be displayed using the **WRITE**, **ZWRITE**, **ZZWRITE**, or **ZZDUMP** commands, as described in Display (Write) Commands.

Refer to the ObjectScript Reference for a list and detailed descriptions of the special variables.

5

Operators and Expressions

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 ultimately evaluated to a value. This topic describes expressions and the operators.

5.1 Introduction to Operators and Expressions

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.

ObjectScript

```
WRITE "1 + 2 * 3 = ", 1 + 2 * 3,!  // returns 9
WRITE "2 * 3 + 1 = ", 2 * 3 + 1,!  // returns 7
WRITE "1 + (2 * 3) = ", 1 + (2 * 3),!  // returns 7
WRITE "2 * (3 + 1) = ", 2 * (3 + 1),!  // returns 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.

ObjectScript

```
WRITE -123 - 3,!       // returns -126
WRITE -123 + -3,!      // returns -126
WRITE -(123 - 3),!     // returns -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 in which ObjectScript performs operations on the expressions. Consider the following expression:

ObjectScript

```
SET TorF = ((4 + 7) > (6 + 6)) // False (0)
WRITE TorF
```

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:

ObjectScript

```
SET Value = (4 + 7 > 6 + 6) // 7
WRITE Value
```

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:

ObjectScript

```
WRITE 1+2*3-4*5,!  // returns 25
WRITE 1+(2*3)-4*5,!  // returns 15
WRITE 1+(2*(3-4))*5,!  // returns -5
WRITE 1+(((2*3)-4)*5),!  // returns 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:

ObjectScript

```
SET x = 3
SET y = 2
IF x && y = 2 {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

This is because 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:

ObjectScript

```
SET x = 3
SET y = 2
IF x && (y = 2) {
  WRITE "True",! }
ELSE {
  WRITE "False",! }
```

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 logical expression:

ObjectScript

```
SET var8=25,var7=23
IF var8 = 25 * (var7 < 24) {
  WRITE !,"True" }
ELSE {
  WRITE !,"False" }
```

ObjectScript evaluates expressions strictly left-to-right. The programmer must use parentheses to establish any precedence. In this case, ObjectScript first evaluates *var8*=25, resulting in 1. It then multiplies this 1 by the results of the expression in parentheses. Because *var7* is less than 24, the expression in parentheses evaluates to 1. Therefore, ObjectScript multiplies 1 * 1, resulting in 1 (true).

5.3 String-to-Number Conversion

A string can be numeric, partially numeric, or non-numeric.

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

5.3.1 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:

ObjectScript

```
WRITE "3" + 4,!             // returns 7
WRITE "003.0" + 4,!         // returns 7
WRITE "++--3" + 4,!         // returns 7
WRITE "3 blind mice" + 4,!  // returns 7
```

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

ObjectScript

```
WRITE "3" < 4,!             // returns 1
WRITE "003.0" < 4,!         // returns 1
WRITE "++--3" < 4,!         // returns 1
WRITE "3 blind mice" < 4,!  // returns 1
```

The following example shows <= comparisons of numeric strings:

ObjectScript

```
WRITE "4" <= 4,!             // returns 1
WRITE "004.0" <= 4,!         // returns 1
WRITE "++--4" <= 4,!         // returns 1
WRITE "4 horsemen" <= 4,!    // returns 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:

ObjectScript

```
WRITE "4" = 4.00,!          // returns 1
WRITE "004.0" = 4,!         // returns 0
WRITE "++--4" = 4,!         // returns 0
WRITE "4 horsemen" = 4,!    // returns 0
WRITE "-4" = -4,!           // returns 1
WRITE "0" = 0,!             // returns 1
WRITE "-0" = 0,!            // returns 0
WRITE "-0" = -0,!           // returns 0
```

5.3.2 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

5.4 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 arrays, operators, or one of the many ObjectScript 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)
```

Expressions can be classified according to what kind of value they return:

- An *arithmetic expression* contains arithmetic operators, gives a numeric interpretation to the operands, and produces a numeric result:

  ### ObjectScript

  ```
  SET expr = 1 + 2
  SET expr = +x
  SET expr = a + b
  ```

  Note that a string used within an arithmetic expression is evaluated as a numeric value (or 0 if it is not a valid numeric value). Also note that using the unary addition operator (+) will implicitly convert a string value to a numeric value.

- A *string expression* contains string operators, gives a string interpretation to the operands, and produces a string result.

  ### ObjectScript

  ```
  SET expr = "hello"
  SET expr = "hello" _ x
  ```

- A *logical expression* contains relational and logical operators, gives a logical interpretation to the operands, and produces a boolean result: TRUE (1) or FALSE (0):

  ### ObjectScript

  ```
  SET expr = 1 && 0
  SET expr = a && b
  SET expr = a > b
  ```

- An *object expression* produces an object reference as a result:

ObjectScript

```
SET expr = object
SET expr = employee.Company
SET expr = ##class(Person).%New()
```

5.4.1 Logical Expressions

Logical expressions use logical operators, numeric relational operators, and string relational operators. They evaluate expressions and result in a Boolean value: 1 (TRUE) or 0 (FALSE). Logical expressions are most commonly used with:

- The IF command
- The $SELECT function
- Postconditional Expressions

In a Boolean test, any expression that evaluates to a non-zero numeric value returns a Boolean 1 (TRUE) value. Any expression that evaluates to a zero numeric value returns a Boolean 0 (FALSE) value. InterSystems IRIS® data platform evaluates a non-numeric string as having a zero numeric value. For further details, refer to String-to-Number Conversion.

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
```

5.5 Arithmetic Operators

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

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:

ObjectScript

```
 WRITE + "32 dollars and 64 cents"        // 32
```

For details, see the Unary Positive (+) reference page.

Unary Negative (-)

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

ObjectScript

```
 SET x = -60
 WRITE " x: ", x,! // -60
 WRITE "-x: ",-x,! // 60
```

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

For details, see the Unary Negative (-) reference page.

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

Addition (+)

The addition operator adds two numeric values. For example:

ObjectScript

```
WRITE 2936.22 + 301.45 //  3237.67
```

For details, see the Addition (+) reference page.

Subtraction (-)

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

ObjectScript

```
WRITE 2936.22 - 301.45 // 2634.77
```

For details, see the Subtraction (-) reference page.

Multiplication (*)

The multiplication operator multiplies two numeric values. For example:

ObjectScript

```
WRITE 9 * 5.5 // 49.5
```

For details, see the Multiplication (*) reference page.

Division (/)

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

ObjectScript

```
WRITE 9 / 5.5 // 1.636363636363636364
```

For details, see the Division (/) reference page.

Integer Division (\ )

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

ObjectScript

```
WRITE "355 \ 113 = ", 355 \ 113 // 3
```

For details, see the Integer Division (\ ) reference page.

Modulo (#)

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

ObjectScript

```
WRITE "37 # 10 = ",37 # 10,! // 7
WRITE "12.5 # 3.2 = ",12.5 # 3.2,! // 2.9
```

For details, see the Modulo (#) reference page.

Exponentiation (**)

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

ObjectScript

```
WRITE "9 ** 2 = ",9 ** 2,! // 81
```

For details, see the Exponentiation (**) reference page. Exponentiation can also be performed using the $ZPOWER function.

Note: InterSystems IRIS supports two representations of numbers: ObjectScript decimal floating-point (referred to as decimal format) and IEEE double-precision binary floating-point (referred to as $DOUBLE, generally used for special purposes). ObjectScript automatically converts a decimal value to the corresponding $DOUBLE value in the following situations:

- If an arithmetic operation involves a $DOUBLE value, ObjectScript converts *all* numbers in the operation to $DOUBLE.
- If an operation results in a number that is too large to be represented in decimal format, ObjectScript automatically converts this number to $DOUBLE, rather than issuing a <MAXNUMBER> error.

For details on these formats, see Numeric Computing in InterSystems Applications.

5.6 Numeric Relational Operators

Numeric relational operators use the numeric values of the operands to produce a Boolean result. When operating on strings, a numeric relational operator treats each of the strings as its numeric value, according to the rules described in String-to-Number Conversion.

Numeric relational operators *should not* be used to compare non-numeric strings.

ObjectScript provides the following numeric relational operators:

Less Than Operator (<)

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

ObjectScript

```
WRITE 9 < 6    // 0
```

For details, see the Less Than (<) reference page.

Greater Than Operator (>)

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

ObjectScript

```
WRITE 15 > 15 // 0
```

For details, see the Greater Than (<) reference page.

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:

ObjectScript

```
WRITE 9 <= 6    // 0
```

See the Less Than or Equal To (<= or '>) reference page.

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:

ObjectScript

```
WRITE 15 >= 15 // 1
```

See the Greater Than or Equal To (<= or '>) reference page.

Note: InterSystems IRIS supports two representations of numbers: ObjectScript decimal floating-point (referred to as decimal format) and IEEE double-precision binary floating-point (referred to as $DOUBLE format, generally used for special purposes).

Less-than/greater-than comparisons between these formats are performed exactly, without rounding. However, equality comparisons between decimal and $DOUBLE numbers often yield unexpected results, and should be avoided. For further details, see Numeric Computing in InterSystems Applications.

5.7 Logical Comparison Operators

The logical comparison operators compare the values of their operands and return a boolean value: TRUE (1) or FALSE (0).

5.7.1 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)
```

5.7.2 Logical Operators

ObjectScript provides the following logical operators:

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, the following statements produce a result of FALSE (0):

ObjectScript

```
SET x=0
WRITE x
```

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).

The following examples evaluate two nonzero-valued operands as TRUE and produces a value of TRUE (1).

ObjectScript

```
SET A=-4,B=1
WRITE A&B // TRUE (1)
```

See the And (& or &&) reference page. Also see the Not And (NAND) ('&) reference page.

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).

The following examples evaluate two TRUE (nonzero) operands, apply the Or to them, and produces a TRUE result:

ObjectScript

```
SET A=5,B=7
WRITE "A!B = ",A!B,!
SET A=5,B=7
WRITE "A||B = ",A||B,!
```

See the Or (! or ||) reference page. Also see the Not Or (NOR) ('!) reference page.

5.8 String Concatenate Operator (_)

The string Concatenate operator (_) interprets its two operands as strings and returns a string value that appends the second string to the first string.

The following example writes the string `Highchair` to the current device.

ObjectScript

```
WRITE "High"_"chair"
```

For details, see the String Concatenate (_) reference page.

5.9 String Relational Operators

String relational operators use the string interpretation of the operands to produce a Boolean result. You can precede any of the string relational operators with the NOT logical operator (') to obtain the negation of the logical result. ObjectScript provides the following string relational operators:

Equals (=)

The Equals operator tests two operands for string equality. When you apply Equals to two strings, ObjectScript returns a result of TRUE (1) if the two operands are identical strings with identical character sequences and no intervening characters, including spaces; otherwise it returns a result of FALSE (0). For example:

ObjectScript

```
WRITE "SEVEN"="SEVEN"    // 1
WRITE "SEVEN"="seven"    // 0
WRITE "SEVEN"=" SEVEN "  // 0
```

For details, see the Equals (=) reference page. Also see the Not Equals ('=) reference page.

Contains ([)

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:

ObjectScript

```
SET L="Steam Locomotive"
SET S="Steam"
WRITE L[S  /// 1
```

For details, see the Contains ([) reference page. Also see the Does Not Contain ('[) reference page.

Follows (])

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:

ObjectScript

```
WRITE "LAMPOON"]"LAMP"  // 1
```

For details, see the Follows (]) reference page. Also see the Not Follows (']) reference page.

Sorts After (]])

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:

ObjectScript

```
WRITE 122]]2  // 1
```

For details, see the Sorts After (]]) reference page. Also see the Not Sorts After (']]) reference page.

5.10 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 if the string *ssn* contains a valid U.S. Social Security Number (3 digits, a hyphen, 2 digits, a hyphen, and 4 digits):

ObjectScript

```
SET ssn="123-45-6789"
SET match = ssn ? 3N1"-"2N1"-"4N
WRITE match
```

For pattern syntax and other details, 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.

5.11 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.

6

Commands

The *command* is the basic unit of code in ObjectScript programming. All of 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.

This topic provides an overview of the most commonly used commands; also see the *ObjectScript Reference*.

6.1 Command Keywords

In ObjectScript all command statements are explicit; every executable line of ObjectScript code must begin with a command keyword. For example, to assign a value to a variable, you must specify the SET keyword, followed by the arguments for the variable and the value to be assigned.

A command always begins with a keyword. Consider the following:

ObjectScript

```
WRITE "Hello"
```

The word WRITE is the command keyword. It specifies the action to perform. The **WRITE** command does exactly what its name implies: it writes to the principal device whatever you specify as its *argument*. In this case, **WRITE** writes the string Hello.

ObjectScript 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.

In an ObjectScript program, the first command on a code line must be indented; the command keyword cannot appear in column 1. When issuing a command from the Terminal command line prompt, or from an **XECUTE** command, no indent is required (indent is permitted).

An executable line of code can contain one or more commands, each with their own command keyword. Multiple commands on a line are separated by one or more spaces. One or more commands may follow a label on the same line; the label and the command are separated by one or more spaces.

In ObjectScript 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.

6.2 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.

6.2.1 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
```

6.2.2 Arguments with Parameters and Postconditionals

Some command arguments accept *parameters* (not to be confused with function parameters). If a given argument can take parameters, the delimiter for the parameters is the colon :).

The following sample command shows the comma used as the argument delimiter and the colon used as the parameter delimiter. In this example, there are two arguments, with four parameters for each argument.

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.

6.2.3 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.

6.2.3.1 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"
  ```

6.3 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**, **THROW**, **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.

6.3.1 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.

6.3.2 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",!
}
```

6.4 Multiple Commands on a Line

A single line of ObjectScript source code may contain multiple commands and their arguments. They 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. Note that if you are using Studio to write or edit source code, this limit may differ.

6.5 Variables

To assign a value to a variable, use the **SET** command.

The **SET** command assigns values to variables. It can assign a value to a single variable or to multiple variables at once.

The most basic syntax of **SET** is:

ObjectScript

```
SET variable = expression
```

This sets the value of a single variable. It also involves several steps:

- ObjectScript evaluates the value expression, determining its value (if possible). This step can generate errors, if the expression contains an undefined variable, invalid syntax (such as division by zero), or other errors.
- If the variable does not already exist, ObjectScript creates it.
- Once the variable has been created, or if it already exists, ObjectScript sets its value to that of the expression.

To set the value for each of multiple variables, use the following syntax:

ObjectScript

```
SET variable1 = expression1, variable2 = expression2, variable3 = expression3
```

To set multiple variables equal to a single expression use the following syntax:

ObjectScript

```
SET (variable1,variable2,variable3)= expression
```

For example, to set the value of the Gender property of an instance of the Person class use the following code:

ObjectScript

```
SET person.Gender = "Female"
```

where *person* is the object reference to the relevant instance of the Person class.

You can also set the Gender property of multiple Person objects at the same time:

ObjectScript

```
SET (per1.Gender, per2.Gender, per3.Gender) = "Male"
```

where *per1*, *per2*, and *per3* are object references to three different instances of the Person class.

You can use **SET** to invoke a method that returns a value. When invoking methods, **SET** allows you to set a variable, global reference, or property equal to the return value of a 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

```
SET retval = ##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 method's return value is assigned to the *retval* local variable. 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

```
SET retval = 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. The method's return value is assigned to the *retval* local variable.

For further details, see SET.

Note: In older code that does not use procedure blocks, the KILL and NEW commands are useful for controlling variable scope. For details, see the reference for those commands.

6.6 Error Processing

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.

6.7 Transaction Processing

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

6.8 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.

6.9 Invoking Code

This section describes commands used for invoking the execution of one or more commands:

- DO
- JOB
- XECUTE
- QUIT and RETURN

6.9.1 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.

6.9.2 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.

6.9.3 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.

6.9.4 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.

6.10 Controlling 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:

- IF, ELSEIF, and ELSE
- FOR
- WHILE and DO WHILE

6.10.1 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 and Expressions.

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:

6.10.1.1 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.

6.10.2 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.

6.10.3 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.

6.11 Controlling I/O

ObjectScript input/output commands provide the basic functionality for getting data in and out of InterSystems IRIS. These are:

- Write Commands
- READ
- OPEN, USE, and CLOSE

6.11.1 Display (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

6.11.1.1 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.

6.11.1.2 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 6–1: 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 6–2: 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: %[illegible]	encoded string	$zwc format with appended /* $bit() */ comment listing 1 bits. For example: %[illegible]	$zwc format with (by default) appended /* $bit() */ comment listing 1 bits. For example: %[illegible]

	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.

6.11.2 READ

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.

6.11.3 OPEN, USE, and CLOSE

For more sophisticated device handling, InterSystems IRIS provides a wealth of options. In short, you can take ownership of an open device with the **OPEN** command; specify the current device with the USE command; and close an open device with the **CLOSE** command. This process as a whole is described in the I/O Device Guide.

For further details, see OPEN, USE, and CLOSE.

7

Callable User-defined Code Modules

This topic describes how to create and invoke user-defined modules of ObjectScript code on InterSystems IRIS® data platform: methods, functions, procedures, routines, or subroutines. The most common form of code is methods, which you define in classes. This topic does not describe methods specifically, but by default, methods are procedures. Because of this, all content on procedures also applies to methods.

As in other languages, ObjectScript allows you to create named code blocks that you can invoke directly. Such blocks are known as procedures. Strictly speaking, in ObjectScript terminology, a code block that is a procedure has a specific syntax and structure.

The syntax of a procedure definition is as follows:

ObjectScript

```
ProcedureName(Parameters) [PublicVariables]
{
    /* code goes here */
    RETURN ReturnValue
}
```

The elements of the procedure, here called **ProcedureName**, are:

- Parameters (zero or more) — These can be of any type and, as is typical of ObjectScript, you do not need to declare their types when you define the procedure. By default, they are passed by value (not by reference). Unless otherwise specified, their scope is local to the procedure. For more information on parameters generally, see Procedure Parameters.
- References to public variables (zero or more) — These, too, can be of any type. The procedure can both reference and set such a variable's value. For more information on public variable references, see Procedure Variables.
- Declaration that the procedure is public (optional) — By default, procedures are private, which means that you can only call them from elsewhere in the same routine (in ObjectScript terminology, a routine is a file containing one or more procedures or other user-defined code blocks). You can also create procedures that are public, using the PUBLIC keyword after the procedure name. Public procedures can be called from other routines or methods. For more information on public and private procedures, see Public and Private Procedures.
- Code — The code in a procedure has all the features available in ObjectScript. Procedure code can also include Java. The code is delimited by curly braces and is also known as a *procedure block*.
- Return value (optional) — This is the value that the procedure returns, and, must be a standard ObjectScript expression. Flow control within a procedure can specify various return values using computed expression values, multiple **RETURN** statements, or both.

Note: Writing procedures is generally preferable to writing subroutines or user-defined functions. Procedure parameters are automatically local in scope within the procedure. They do not require a **NEW** command to ensure that they do not overwrite other values, since they are private to the procedure and do not interact with the symbol table. Also, the explicit declaration of public variables allows you to refer to global variables within an application, such a bank-wide interest rate; it also allows you to create and set values for variables within the procedure that are available to the rest of an application.

Procedures are a particular kind of ObjectScript routine.

InterSystems IRIS also provides a large number of system-supplied functions, all of which are described in the ObjectScript Language Reference; these are sometimes known as intrinsic functions. Calls to system functions are identified by a $ prefix.

7.1 Procedures, Routines, Subroutines, Functions, and Methods: What Are They?

This topic describes how to implement your own code using procedures, which are the recommended form for implementing user-defined functionality. InterSystems documentation describes procedures, routines, subroutines, functions, and methods. Though all these entities share features, each has its own characteristics.

The most flexible, most powerful, and recommended form of named, user-defined code block is the procedure. The features of a procedure includes that it:

- Can be private or public.
- Can accept zero or more parameters.
- Automatically maintains any variables created within it as local in scope.
- Can refer to and alter variables outside it.
- Can return a value of any type or no return value.

By contrast:

- A subroutine is always public and cannot return a value.
- A function is always public, requires explicit declaration of local variables (and, otherwise, overwrites external variables), and must have a return value.
- By default, a method is a procedure that is specified as part of a class definition and that you can invoke on one or more objects or on a class. If you explicitly declare it a function, it is then a function with all the accompanying characteristics; this is not recommended.
- A routine is an ObjectScript program. It can include one or more procedures, subroutines, and functions, as well as any combination of the three.

Note: ObjectScript also supports a related form of user-defined code through its macro facility.

7.1.1 Routines

A routine is a callable block of user-written code that is an ObjectScript program. A routine performs commonly needed operations. Its name is determined by the name of the .MAC file that you choose for saving it. Depending on if a routine returns a value, you can invoke a routine with one or both of the following sets of syntax:

ObjectScript

```
DO ^RoutineName
SET x = $$^RoutineName
```

A routine is defined within a namespace. You can use an extended routine reference to execute a user-defined routine defined in a namespace other than the current namespace:

ObjectScript

```
DO ^|"USER"|RoutineName
```

Generally, routines serve as containers for subroutines, methods, and procedures.

The routine is identified by a label (also referred to as a tag) at the beginning of the block of code. This label is the name of the routine. This label is (usually) followed by parentheses which contain a list of parameters to be passed from the calling program to the routine.

When you save a routine to a file, the file name cannot include the underscore (_), hyphen (-), or semicolon (;) characters; names that include such characters are not valid.

7.1.2 Subroutines

A subroutine is a named block of code within a routine. Typically, a subroutine begins with label and ends with a **QUIT** statement. It can accept parameters and does not return a value. To invoke a subroutine, use the following syntax:

```
DO Subroutine^Routine
```

where *Subroutine* is a code block within the *Routine* file (Routine.MAC).

The form of a subroutine is:

ObjectScript

```
Label(parameters) // comment
 // code
 QUIT // note that QUIT has no arguments
```

For more details on subroutines, see the section below on subroutines as legacy code.

If you enclose the code and **QUIT** statement within curly braces, the subroutine is a procedure and can be treated as such. In such a case, a **QUIT** statement is redundant and can be omitted. For example, the following two subroutine definitions are equivalent:

ObjectScript

```
Label(parameters) PUBLIC {
 // code
 QUIT
}
```

And:

ObjectScript

```
Label(parameters) PUBLIC {
 // code
}
```

7.1.3 Functions

InterSystems IRIS comes with many system-supplied functions (sometimes known as *intrinsic functions*), which are described in the ObjectScript Language Reference. This section describes user-defined (*extrinsic*) functions.

A function is a named block of code within a routine. Typically, a function begins with label and ends with a **RETURN** statement. It can accept parameters and can also return a value. To invoke a function, there are two valid forms of the syntax:

ObjectScript

```
 SET rval=$$Function()  /* returning a value */
 DO Function^Routine   /* ignoring the return value */
```

where *Function* is a code block within the *Routine* file (Routine.MAC). In both syntactic forms you can use an extended routine reference to execute a function located in a different namespace.

The form of a function is:

ObjectScript

```
Label(parameters)
 // code
 RETURN ReturnValue
```

If you enclose the code and its **RETURN** statement within curly braces, the function is a procedure and can be treated as such. Note that because a procedure is private by default, you may wish to specify the PUBLIC keyword, as follows:

ObjectScript

```
Label(parameters) PUBLIC {
 // code
 RETURN ReturnValue }
```

The following example defines a simple function (MyFunc) and calls it, passing two parameters and receiving a return value:

ObjectScript

```
Main ;
   TRY {
     KILL x
     SET x=$$MyFunc(7,10)
        WRITE "returned value is ",x,!
        RETURN
   }
      CATCH { WRITE $ZERROR,!
   }
MyFunc(a,b)
   SET c=a+b
   RETURN c
```

The code invoking the function can ignore the return value, but a function's **RETURN** command must specify a return value. Attempting to exit a function with an argumentless **RETURN** generates a <COMMAND> error. The <COMMAND> error specifies the location of the call that invoked the function, followed by a message that specifies the offset location of the argumentless **RETURN** command within the called function. Refer to $ZERROR for further details.

For more details on functions, see the section below on functions as legacy code.

7.2 Defining Procedures

As in other languages, a procedure is a series of ObjectScript commands (a section of a larger routine) that accomplishes a specific task. Similar to constructs such as **If**, the code of a procedure is contained within curly braces.

Procedures allow you to define each variable as either public or private. For example, the following procedure, is called `MyProc`:

ObjectScript

```
MyProc(x,y) [a,b] PUBLIC {
 Write "x + y = ", x + y
}
```

defines a public procedure named `MyProc` which takes two parameters, *x* and *y*. It defines two public variables, *a* and *b*. All other variables used in the procedure (in this case, *x* and *y*) are private variables.

By default, procedures are private, which means that you can only call them from elsewhere in the same routine. You can also create procedures that are public, using the Public keyword after the procedure name. Public procedures can be called from other routines.

Procedures do not need to have defined parameters. To create procedures with parameters, place a parenthesized list of variables immediately after the label.

7.2.1 Invoking Procedures

To invoke a procedure, either issue a **DO** command that specifies the procedure, or call it as a function using the $$ syntax. You can control whether a procedure can be invoked from any program (public), or only from the program in which it is located (private). If invoked with **DO**, a procedure does not return a value; if invoked as a function call, a procedure returns a value. The $$ form provides the most functionality, and is generally the preferred form.

7.2.1.1 Using the $$ Prefix

You can invoke a user-defined function in any context in which an expression is allowed. A user-defined function call takes the form:

```
$$name([param[ ,...]])
```

where:

- *name* specifies the name of the function. Depending on where the function is defined, name can be specified as:
  - *label* is a line label within the current routine.
  - *label^routine* is a line label within the named routine that resides on disk.
  - *^routine* is a routine that resides on disk. The routine must contain only the code for the function to be performed.
- *param* specifies the values to be passed to the function. The supplied parameters are known as the *actual parameter list*. They must match the *formal parameter list* defined for the function. For example, the function code may expect two parameters, with the first being a numeric value and the second being a string literal. If you specify the string literal for the first parameter and the numeric value for the second, the function may yield an incorrect value or possibly generate an error. Parameters in the formal parameter list always have **NEW** invoked by the function. See the NEW command. Parameters can be passed by value or by reference. See Parameter Passing. If you pass fewer parameters to the function than are listed in the function's formal parameter list, parameter defaults are used (if defined); if there are no defaults, these parameters remain undefined.

7.2.1.2 Using the DO Command

You can invoke a user-defined function using the **DO** command. (You cannot invoke a system-supplied function using the **DO** command.) A function invoked using **DO** does not return a value. That is, the function must generate a return value, but the **DO** command ignores this return value. This greatly limits the use of **DO** for invoking user-defined functions.

To invoke a user-defined function using **DO**, you issue a command in the following syntax:

DO label(*param*[,...])

The **DO** command calls the function named *label* and passes it the parameters (if any) specified by *param*. Note that the $$ prefix is not used, and that the parameter parentheses are mandatory. The same rules apply for specifying the *label* and *param* as when invoking a user-defined function using the $$ prefix.

A function must always return a value. However, when a function is called with **DO**, this returned value is ignored by the calling program.

7.2.2 Procedure Syntax

Procedure syntax:

```
label([param[=default]][,...]) [[pubvar[,...]]] [access]
 {
  code
 }
```

Invoking syntax:

```
DO label([param[,...]])
```

or

```
command $$label([param][,...])
```

where

Argument	Description
label	The procedure name. 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 procedure. These expected parameters are known as the *formal parameter list*. The parameters themselves are optional (there may be none, one, or more than one *param*) but the parentheses are mandatory. Multiple *param* values are separated by commas. Parameters may be passed to the formal parameter list by value or by reference. Procedures that are routines do not include type information about their parameters; procedures that are methods do include this information. The maximum number of formal parameters is 255.
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
pubvar	Public variables. An optional 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. Multiple *pubvar* values are separated by commas. All variables not declared as public variables are private variables. Private variables are available only to the current invocation of the procedure. They are undefined when the procedure is invoked, and destroyed when the procedure is exited. If the procedure calls any code outside of that procedure, the private variables are preserved, but are unavailable until the call returns to the procedure. All % variables are always public, whether or not they are listed here. The list of public variables can include one or more of the *param* specified for this routine.
access	An optional keyword that declares whether the procedure is public or private. There are two available values: 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.
code	A block of code, enclosed in curly braces. 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.

You cannot insert a line break between a command and its arguments.

Each procedure is implemented as part of a routine; each routine can contain multiple procedures.

In addition to standard ObjectScript syntax, there are special rules governing routines. A line in a routine can have a label at the beginning (also called a tag), ObjectScript code, and a comment at the end; but all of these elements are optional.

InterSystems recommends that the first line of a routine have a label matching the name of the routine, followed by a tab or space, followed by a short comment explaining the purpose of the routine. If a line has a label, you must separate it from the rest of the line with a tab or a space. This means that as you add lines to your routine using Studio, you either type a label and a tab/space, followed by ObjectScript code, or you skip the label and type a tab or space, followed by ObjectScript. So in either case, every line must have a tab or space before the first command.

To denote a single-line comment use either a double slash (`//`) or a semicolon (`;`). If a comment follows code, there must be a space before the slashes or semicolon; if the line contains only a comment, there must be a tab or space before the slashes or semicolon. By definition, there can be no line break within a single-line comment; for a multiline comment, mark the beginning of the comment with `/*` and the end with `*/`.

7.2.3 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. Hence, you do not have to declare them as such and they do not require a **NEW** command. To share some of these variables with procedures that this procedure calls, pass them as parameters to the other procedures.

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>
```

7.2.3.1 Public versus Private Variables

Within a procedure, local variables may be either *public* or *private*. The public list [*pubvar*] declares which variable references in the procedure are added to the set of public variables; all other variable references in the procedure are to a private set seen only by the current invocation of the procedure.

Private variables are undefined when a procedure is entered, and they are destroyed when a procedure is exited.

When code within a procedure calls any code outside of that procedure, the private variables are restored upon the return to the procedure. The called procedure or routine has access to public variables (as well as its own private ones.) Thus, [*pubvar*] specifies both the public variables seen by this procedure and the variables used in this procedure that are capable of being seen by a routine that the procedure calls.

If the public list is empty, then all variables are private. In this case, the square brackets are optional.

Variables whose name starts with the percent (%) character are typically variables used by the system or for some special purpose. InterSystems IRIS reserves all % variables (except %z and %Z variables) for system use; user code should only use % variables that begin with %z or %Z. All % variables are implicitly public. They can be listed in the public list (for documentation purposes) but this is not necessary.

7.2.3.2 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.

7.2.3.3 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.

7.2.4 Public and Private Procedures

A procedure can be public or private. A private procedure can only be called from within the routine in which the procedure is defined, whereas a public procedure can be called from any routine. If the PUBLIC and PRIVATE keywords are omitted, the default is private.

For instance,

ObjectScript

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

defines a public procedure, while

ObjectScript

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

and

ObjectScript

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

both define a private procedure.

7.3 Parameter Passing

An important features of procedures is their support for parameter passing. This is the mechanism by which you can pass values (or variables) to a procedure as parameters. Of course, parameter passing is not required; for example, procedures with no parameter passing could be used to generate a random number or to return the system date in a format other than the default format. Commonly, however, procedures do use parameter passing.

To set up parameter passing, specify:

- An *actual parameter list* on the procedure call.
- A *formal parameter list* on the procedure definition.

When InterSystems IRIS executes a user-defined procedure, it maps the parameters in the actual list, by position, to the corresponding parameters in the formal list. Thus, the value of the first parameter 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 parameters is done by position, not name. Thus, the variables used for the actual parameters and the formal parameters are not required to have (and usually should not have) the same names. The procedure accesses the passed values by referencing the appropriate variables in its formal list.

The actual parameter list and the formal parameter list may differ in the number of parameters:

- If the actual parameter list has fewer parameters than the formal parameter list, the unmatched elements in the formal parameter list are undefined. You can specify a default value for an undefined formal parameter, as shown in the following example:

  ObjectScript

  ```
  Main
     /* Passes 2 parameters to a procedure that takes 3 parameters */
     SET a="apple",b="banana",c="carrot",d="dill"
     DO ListGroceries(a,b)
     WRITE !,"all done"
  ListGroceries(x="?",y="?",z="?") {
     WRITE x," ",y," ",z,!   }
  ```

- If the actual parameter list has more parameters than the formal parameter list, a <PARAMETER> error occurs, as shown in the following example:

  ObjectScript

  ```
  Main
     /* Passes 4 parameters to a procedure that takes 3 parameters.
        This results in a <PARAMETER> error  */
     SET a="apple",b="banana",c="carrot",d="dill"
     DO ListGroceries(a,b,c,d)
     WRITE !,"all done"
  ListGroceries(x="?",y="?",z="?") {
     WRITE x," ",y," ",z,!   }
  ```

If there are more variables in the formal list than there are parameters in the actual list, and a default value is not provided for each, the extra variables are left undefined. Your procedure code should include appropriate **IF** tests to make sure that each procedure reference provides usable values. To simplify matching the number of actual parameters and formal parameters, you can specify a variable number of parameters.

The maximum number of actual parameters is 254.

When passing parameters to a user-defined procedure, you can use passing by value or passing by reference. You can mix passing by value and passing by reference within the same procedure call.

- Procedures can be passed parameters by value or by reference.

- Subroutines can be passed parameters by value or by reference.
- User-defined functions can be passed parameters by value or by reference.
- System-supplied functions can be passed parameters by value only.

7.3.1 Passing By Value

To pass by value, specify a literal value, an expression, or a local variable (subscripted or unsubscripted) in the actual parameter list. In the case of an expression, InterSystems IRIS first evaluates the expression and then passes the resulting value. In the case of a local variable, InterSystems IRIS passes the variable's current value. Note that all specified variable names must exist and must have a value.

The procedure's formal parameter list contains unsubscripted local variable names. It cannot specify an explicit subscripted variable. However, specifying a variable number of parameters implicitly creates subscripted variables.

InterSystems IRIS implicitly creates and declares any non-public variables used within a procedure, so that already-existing variables with the same name in calling code are not overwritten. It places the existing values for these variables (if any) on the program stack. When it invokes the **QUIT** or **RETURN** command, InterSystems IRIS executes an implicit **KILL** command for each of the formal variables and restores their previous values from the stack.

In the following example, the **SET** commands use three different forms to pass the same value to the referenced **Cube** procedure.

ObjectScript

```
 DO Start()
 WRITE "all done"
Start() PUBLIC {
 SET var1=6
 SET a=$$Cube(6)
 SET b=$$Cube(2*3)
 SET c=$$Cube(var1)
 WRITE !,"a: ",a," b: ",b," c: ",c,!
 RETURN 1
 }
Cube(num) PUBLIC {
 SET result = num*num*num
 RETURN result
}
```

7.3.2 Passing By Reference

To pass by reference, specify a local variable name or the name of an unsubscripted array in the actual parameter list, using the form:

```
.name
```

With passing by reference, a specified variable or array name does not have to exist before the procedure reference. You typically pass arrays by reference only. You cannot pass a subscripted variable by reference.

- Actual parameter list: The period preceding the local variable or array name in the actual parameter list is required. It specifies that the variable is being passed by reference, not passed by value.
- Formal parameter list: No special syntax is required in the formal parameter list to receive a variable passed by reference. The period prefix is not permitted in the formal parameter list. However, an ampersand (&) prefix is permitted before the name of a variable in the formal parameter list; by convention this & prefix is used to indicate that this variable is being passed in by reference. The & prefix is optional and performs no operation; it is a useful convention for making your source code easier to read and maintain.

A method definition can specify passing by reference, as described in Indicating How Arguments Are to Be Passed.

In passing by reference, each variable or array name in the actual list is *bound* to the corresponding variable name in the function's formal list. Passing by reference provides an effective mechanism for two-way communication between the referencing routine and the function. Any change that the function makes to a variable in its formal list is also made to the corresponding by-reference variable in the actual list. This also applies to the **KILL** command. If a by-reference variable in the formal list is killed by the function, the corresponding variable in the actual list is also killed.

If a variable or array name specified in the actual list does not already exist, the function reference treats it as undefined. If the function assigns a value to the corresponding variable in the formal list, the actual variable or array is also defined with this value.

The following example compares passing by reference with passing by value. The variable *a* is passed by value, the variable *b* is passed by reference:

ObjectScript

```
Main
   SET a="6",b="7"
     WRITE "Initial values:",!
     WRITE "a=",a," b=",b,!
   DO DoubleNums(a,.b)
     WRITE "Returned to Main:",!
     WRITE "a=",a," b=",b
DoubleNums(foo,&bar) {
  WRITE "Doubled Numbers:",!
  SET foo=foo*2
  SET bar=bar*2
  WRITE "foo=",foo," bar=",bar,!
  }
```

The following example uses passing by reference to achieve two-way communication between the referencing routine and the function through the variable *result*. The period prefix specifies that *result* is passed by reference. When the function is executed, *result* in the actual parameter list is created and bound to *z* in the function's formal parameter list. The calculated value is assigned to *z* and passed back to the referencing routine in *result*. The & prefix to *z* in the formal parameter list is optional and non-functional, but helps to clarify the source code. Note that *num* and *powr* are passed by value, not reference. This is an example of mixing passing by value and passing by reference:

ObjectScript

```
Start ; Raise an integer to a power.
 READ !,"Integer= ",num  RETURN:num=""
 READ !,"Power= ",powr   RETURN:powr=""
 SET output=$$Expo(num,powr,.result)
 WRITE !,"Result= ",output
 GOTO Start
Expo(x,y,&z)
 SET z=x
 FOR i=1:1:y {SET z=z*x}
 RETURN z
```

7.3.3 Variable Number of Parameters

A procedure can specify that it accepts a variable number of parameters. You do this by appending three dots to the name of the final parameter; for example, `vals...`. This parameter must be the final parameter in the parameter list. It can be the only parameter in the parameter list. This `...` syntax can pass multiple parameters, a single parameter, or zero parameters.

Spaces and new lines are permitted between parameters in the list, as well as before the first parameter and after the final parameter in the list. Whitespace is not permitted between the three dots.

To use this syntax, specify a signature where the name of the final parameter is followed by `...`. The multiple parameters passed to the method through this mechanism can have values from data types, be object-valued, or be a mix of the two. The parameter that specifies the use of a variable number of parameters can have any valid identifier name.

ObjectScript handles passing a variable number of parameters by creating a subscripted variable, creating one subscript for each passed variable. The top level of the variable contains the number of parameters passed. The variable subscripts contain the passed values.

This is shown in the following example. It uses *invals...* as the only parameter in the formal parameter list. ListGroceries(invals...) receives a variable number of values passed by value:

ObjectScript

```
Main
   SET a="apple",b="banana",c="carrot",d="dill",e="endive"
   DO ListGroceries(a,b,c,d,e)
   WRITE !,"all done"
ListGroceries(invals...) {
  WRITE invals," parameters passed",!
   FOR i=1:1:invals {
      WRITE invals(i),! }
   }
```

The following example uses *morenums...* as the final parameter, following two defined parameters. This procedure can receive a variable number of additional parameters, starting with the third parameter. The first two parameters are required, either as defined parameters `DO AddNumbers(a,b,c,d,e)` or as placeholder commas `DO AddNumbers(,,c,d,e)`:

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(a,b,c,d,e)
   WRITE "all done"
AddNumbers(x,y,morenums...) {
  SET sum = x+y
  FOR i=1:1:$GET(morenums, 0) {
     SET sum = sum + $GET(morenums(i)) }
  WRITE "The sum is ",sum,!
  }
```

The following example uses *morenums...* as the final parameter, following two defined parameters. This procedure receives exactly two parameter values; the *morenums...* variable number of additional parameters is 0:

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(a,b)
   WRITE "all done"
AddNumbers(x,y,morenums...) {
  SET sum = x+y
  FOR i=1:1:$GET(morenums, 0) {
     SET sum = sum + $GET(morenums(i)) }
  WRITE "The sum is ",sum,!
  }
```

As specified, `AddNumbers(x,y,morenums...)` can receive a minimum of two parameters and a maximum of 255. If you supply defaults for the defined parameters `AddNumbers(x=0,y=0,morenums...)` this procedure can receive a minimum of no parameters and a maximum of 255.

The following example uses *nums...* as the only parameter. It receives a variable number of values passed by reference:

ObjectScript

```
Main
   SET a=7,b=8,c=9,d=100,e=2000
   DO AddNumbers(.a,.b,.c,.d,.e)
   WRITE "all done"
AddNumbers(&nums...) {
  SET sum = 0
  FOR i=1:1:$GET(nums, 0) {
     SET sum = sum + $GET(nums(i)) }
  WRITE "The sum is ",sum,!
  RETURN sum
  }
```

When a variable parameter list params... receives parameters passed by reference and passes the params... to a routine, the intermediate routine can add additional parameters (additional nodes in the params array) that will also be passed by reference. This is shown in the following example:

ObjectScript

```
Main
   SET a(1)=10,a(2)=20,a(3)=30
   DO MoreNumbers(.a)
   WRITE !,"all done"
MoreNumbers(&params...) {
   SET params(1,6)=60
   SET params(1,8)=80
   DO ShowNumbers(.params) }
ShowNumbers(&tens...) {
   SET key=$ORDER(tens(1,1,""),1,targ)
   WHILE key'="" {
     WRITE key," = ",targ,!
     SET key=$ORDER(tens(1,1,key),1,targ)
   }
  }
```

The following example shows that this *args...* syntax can be used in both the formal parameter list and in the actual parameter list. In this example, a variable number of parameters (*invals...*) are passed by value to ListNums, which doubles their values then passes them as *invals...* to ListDoubleNums:

ObjectScript

```
Main
   SET a="1",b="2",c="3",d="4"
   DO ListNums(a,b,c,d)
   WRITE !,"back to Main, all done"
ListNums(invals...) {
  FOR i=1:1:invals {
    WRITE invals(i),!
    SET invals(i)=invals(i)*2 }
  DO ListDoubleNums(invals...)
  WRITE "back to ListNums",!
}
ListDoubleNums(twicevals...) {
  WRITE "Doubled Numbers:",!
  FOR i=1:1:twicevals {
    WRITE twicevals(i),! }
}
QUIT
```

Also see Variable Numbers of Arguments in Methods.

7.4 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.

7.5 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.

7.6 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.

7.7 Legacy User-Defined Code

Before the addition of procedures to InterSystems IRIS, there was support for user-defined code in the form of subroutines and functions (which themselves can now be implemented as procedures). These legacy entities are described here, primarily to help explicate already-written code; their ongoing use is not recommended.

7.7.1 Subroutines

7.7.1.1 Syntax

Routine 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 to the formal parameter list 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.

7.7.1.2 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 Parameter Passing.

The same variables are available to a subroutine and its calling routine.

A subroutine does not return a value.

7.7.2 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.

7.7.2.1 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 to the formal parameter list 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.

7.7.2.2 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()
```

7.7.2.3 Function Parameters

As a rule, user-defined functions use parameter passing. A function, however, can work without externally supplied values. For example, you can define a function to generate a random number or to return the system date in a format other than the default format. Note that in these cases, too, you must supply the parameter parentheses in both the function definition and the function call, even though the parameter list is empty.

Parameter passing requires:

- An *actual parameter list* on the function call.
- A *formal parameter list* on the function definition.

When InterSystems IRIS executes a user-defined function, it maps the parameters in the actual list, by position, to the corresponding parameters in the formal list. For example, the value of the first parameter 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 parameters is done by position, not name. Thus, the variables used for the actual parameters and the formal parameters are not required to have (and usually should not have) the same names. The function accesses the passed values by referencing the appropriate variables in its formal list.

If there are more variables in the formal list than there are parameters in the actual list, and a default value is not provided for each, the extra variables are left undefined. Your function code should include appropriate **If** tests to make sure that each function reference provides usable values.

When passing parameters to a user-defined function, you can use passing by value or passing by reference. You can mix passing by value and passing by reference within the same function call. See Parameter Passing.

7.7.2.4 Return Value

The syntax for defining a user-defined function is as follows:

```
label ( parameters  )
   code
   QUIT expression
```

The function must contain a **QUIT** command followed by an expression. InterSystems IRIS terminates the execution of the function when it encounters the **QUIT**, and returns the single value that results from the associated expression to the invoking program.

If you specify a **QUIT** command without an expression, InterSystems IRIS issues an error.

7.7.2.5 Variables

The invoking program and the called function use the same set of variables, with the following special considerations.

- InterSystems IRIS executes an implicit **NEW** command for each parameter in the formal list. This is shown in the following example, where x is reinitialized when myfunc is invoked:

  ObjectScript

  ```
  mainprog
   SET x=7
   SET y=$$myfunc(99)
  myfunc(x)
   WRITE x
   QUIT 66
  ```

- The system saves the current value of the system variable **$TEST** when it enters the function and restores it when the function terminates. Any change in the **$TEST** value during execution of the function will be discarded when the function exits, unless you include code to explicitly save it by some other means.

7.7.2.6 Location of Functions

You can define a user-defined function within the routine that references it, or in a separate routine where multiple programs can reference it. Recommended practice is to use one routine to contain all your user-defined function definitions. In this way, you can easily locate any function definition to examine or update it.

7.7.2.7 Invoking a User-defined Function

You can invoke a user-defined function using either the $$ prefix, or by using the **DO** command. The $$ form provides the most functionality, and is generally the preferred form.

Using the $$ Prefix

You can invoke a user-defined function in any context in which an expression is allowed. A user-defined function call takes the form:

```
$$name([param [,...]])
```

where

- *name* specifies the name of the function. Depending on where the function is defined, name can be specified as:
  - *label* — A line label within the current routine.
  - *label^routine* — A line label within the named routine that resides on disk.
  - *^routine* — A routine that resides on disk. The routine must contain only the code for the function to be performed.

  A routine is defined within a namespace. You can use an extended routine reference to execute a user-defined function that is located in a routine defined in a namespace other than the current namespace:

  ObjectScript

  ```
  WRITE $$myfunc^|"USER"|routine
  ```

- *param* specifies the values to be passed to the function. The supplied parameters are known as the *actual parameter list*. They must match the *formal parameter list* defined for the function. For example, the function code may expect two parameters, with the first being a numeric value and the second being a string literal. If you specify the string literal for the first parameter and the numeric value for the second, the function may yield an incorrect value or possibly generate an error. Parameters in the formal parameter list always have **NEW** invoked by the function. See the NEW command. Parameters can be passed by value or by reference. See Parameter Passing. If you pass fewer parameters to the function than are listed in the function's formal parameter list, parameter defaults are used (if defined); if there are no defaults, these parameters remain undefined.

Using the DO Command

You can invoke a user-defined function using the **DO** command. (You cannot invoke a system-supplied function using the **DO** command.) A function invoked using **DO** does not return a value. That is, the function must generate a return value, but the **DO** command ignores this return value. This greatly limits the use of **DO** for invoking user-defined functions.

To invoke a user-defined function using **DO**, you issue a command in the following syntax:

```
DO label(param[ ,...])
```

The **DO** command calls the function named *label* and passes it the parameters (if any) specified by *param*. Note that the $$ prefix is not used, and that the parameter parentheses are mandatory. The same rules apply for specifying the *label* and *param* as when invoking a user-defined function using the $$ prefix.

A function must always return a value. However, when a function is called with **DO**, this returned value is ignored by the calling program.

8

Using 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 file (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.

A suitable IDE will provide 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.

8.1 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.

8.2 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.

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 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.

8.3.5 Macro Comments and Studio Assist

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 wish to use Studio Assist with 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 Studio Assist popup. (All macros in the current file appear in the Studio Assist popup.) For example, if the following code were referenced through an **#include** directive, then the first macro would appear in the Studio Assist popup and the second would not:

ObjectScript

```
/// A macro that is visible with Studio Assist
#define MyAssistMacro 100
 //
 // ...
 //
 // A macro that is not visible with Studio Assist
#define MyOtherMacro -100
```

For information on making macros available through include files, see Including Include Files.

8.4 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.

8.5 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.

8.6 See Also

- #define
- #include
- Preprocessor Directives Reference
- System Macros

9

Embedded SQL

You can embed SQL within ObjectScript.

9.1 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.

9.2 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.

10

Multidimensional Arrays

InterSystems IRIS® data platform includes support for multidimensional arrays. A multidimensional array is a persistent variable consisting of one or more elements, each of which has a unique subscript. You can intermix different kinds of subscripts. An example is the following *MyVar* array:

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

The array node *MyVar* is an ObjectScript variable and follows the conventions for that variable type.

The subscripts of *MyVar* are positive and negative numbers, strings, and combinations of these. A subscript can include any characters, including Unicode characters. A numeric subscript is stored and referenced as a canonical number. A string subscript is stored and referenced as a case-sensitive literal. A canonical number (or a number that reduces to a canonical number) and a string containing that canonical number are equivalent subscripts.

10.1 What Multidimensional Arrays Are

Succinctly, multidimensional arrays are persistent, n-dimensional arrays that are denoted through the use of subscripts. Individual nodes are also known as “globals” and are the building block of InterSystems IRIS data storage. They have other characteristics as well:

- They exist in tree structures.
- They are sparse.
- They are one of three basic kinds.

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.

10.1.3 Kinds of Multidimensional Arrays

Multidimensional arrays can be one of three basic kinds. In all of these cases, an array can be differentiated from a scalar by virtue of the fact that an array has one or more subscripts.

- Any local variable with subscripts is an array. For example, creating *x(1)* defines *x* as an array.
- Any global with subscripts is an array. For example, creating *^y(1)* defines *^y* as an array.
- 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 ];
  ```

  You can then set its value with a statement such as:

  ```
  myObj.MyProp(1) = "hello world"
  ```

  Note that multidimensional properties in persistent or serial classes are not saved to disk when an object is saved, unless custom code is written to save the property.

10.2 Manipulating Multidimensional Arrays

You can write to and read from them using the **Read** and **Write** commands respectively.

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.

This set of commands and functions can operate on multidimensional globals and multidimensional local variables. Globals can be easily identified by their leading "^" (caret) character.

10.3 See Also

For further information on multidimensional arrays, see Using Globals.

11

String Operations

ObjectScript provides several groups of operations related to strings, each with its own purpose and features. These are basic string operations and functions; delimited string operations; and list-structure string operations.

11.1 Basic String Operations and Functions

ObjectScript basic string operations allow you to perform various manipulations on a string. They 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. For more details, see $LENGTH.

- **$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
  ```

  For more details, see $JUSTIFY.

- **$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
  ```

For more details, see $ZCONVERT.

- 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." For more details, see $FIND.

- 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. For more details, see $TRANSLATE.

- 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. For more details, see $REPLACE.

- 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
  ```

  For more details, see the next section or see $EXTRACT.

11.1.1 Advanced Features of $EXTRACT

You can use the **$EXTRACT** function in conjunction with the **SET** command pad a string on the left with spaces.

ObjectScript

```
SET x = "abc"
WRITE x,!
SET $EXTRACT(y, 3) = x
SET x = y
WRITE x
```

This code takes the string `abc` and places at the third character of string *y*. Because *y* has no specified value, **$EXTRACT** assumes that its characters are blank, which acts to pad the string.

You can also use **$EXTRACT** to insert a new string at a particular point in variable. It extracts the characters specified and replaces them with the supplied substring, whether or not the lengths of the old and new strings match. For example:

ObjectScript

```
SET x = "1234"
WRITE x,!
SET $EXTRACT(x, 3) = "abc"
WRITE x,!
SET $EXTRACT(y, 3) = "abc"
WRITE y
```

This code sets *x* to `1234`; it then extracts the third character of *x* using **$EXTRACT** and inserts `abc` in its place, making the string `12abc4`.

11.2 Delimited Strings

ObjectScript includes functions that allow you to work with strings as a set of substrings, so that you can pass related pieces of data as a single string. These are functions are:

- **$PIECE** — Returns a specific piece of a string based on a specified delimiter. It can also return a range of pieces, as well as multiple pieces from a single string, based on multiple delimiters.
- **$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.

11.2.1 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.

11.3 List-Structure String Operations

ObjectScript defines a special kind of string called a *list*, which consists of an encoded list of substrings, known as elements. These InterSystems IRIS 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*.

11.3.1 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.)

11.4 Lists and Delimited Strings Compared

11.4.1 Advantages of Lists

- Lists do 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**.

11.4.2 Advantages of Delimited Strings

- 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 an InterSystems IRIS list is an encoded string, you can only use $List functions on an InterSystems IRIS list.

12

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.

12.1 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.

12.2 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 `^sample.person(id)`.

In contrast, InterSystems IRIS is not optimized for lock names such as `^name_concatenated_identifier`. Non-subscripted lock names can also cause performance problems related to ECP.

12.3 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:

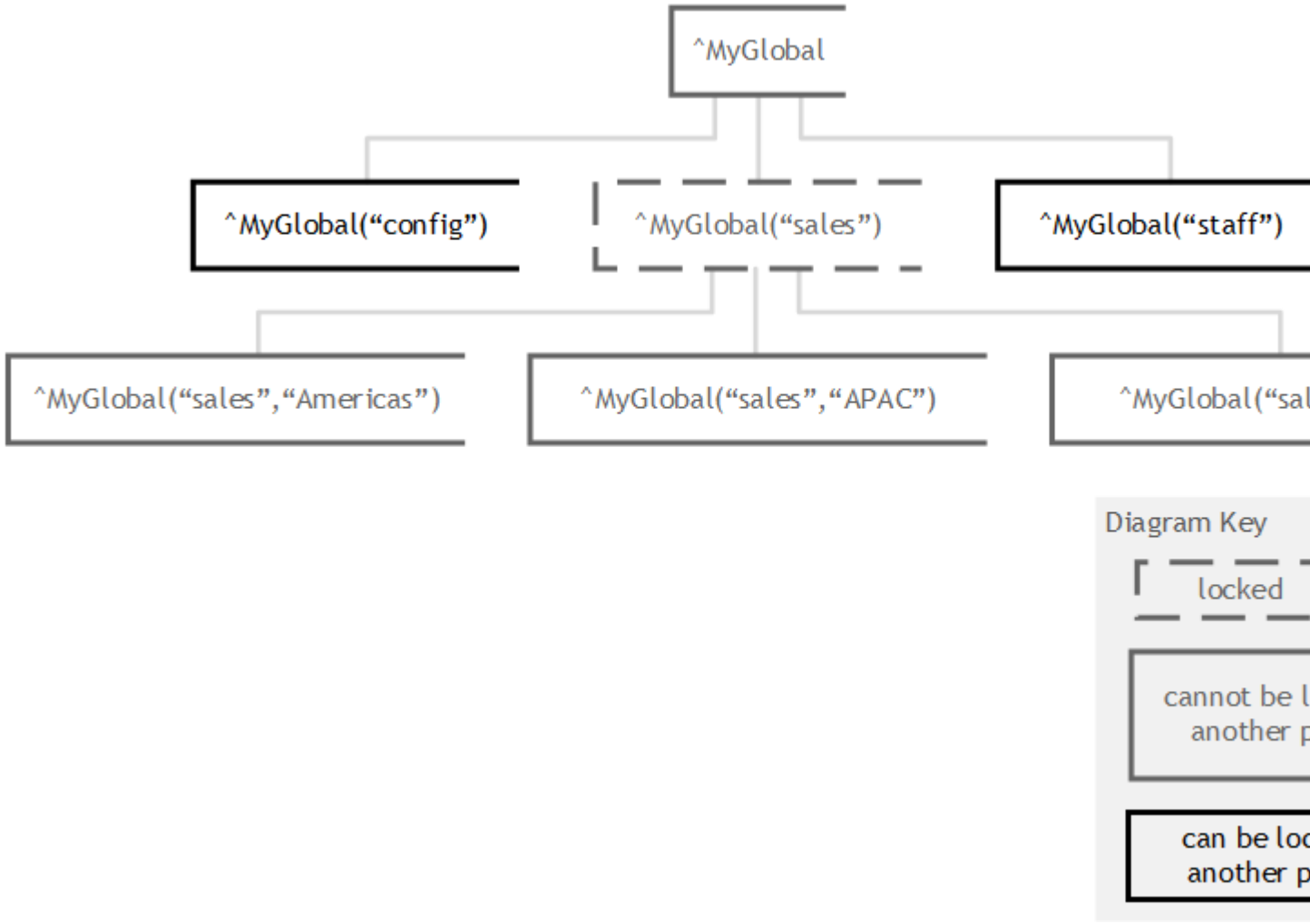
```
LOCK TABLE FULL
```

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.)

12.4 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.

12.5 Using the LOCK Command

This section discusses how to use the LOCK command to add and remove locks.

12.5.1 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.

12.5.2 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)}
```

12.5.2.1 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.

12.5.3 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.

12.5.4 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.

12.5.4.1 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 (-).

12.5.4.2 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.

12.6 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.

12.6.1 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.

12.6.2 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.

12.6.3 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.

12.7 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.

12.7.1 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.

12.7.2 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.

12.8 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.

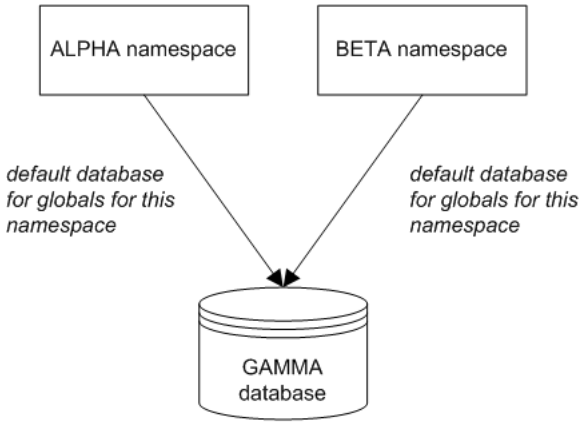
Although lock names are intrinsically arbitrary, when you use a lock name that starts with a caret (^), InterSystems IRIS provides special behavior appropriate for these scenarios. The following subsections give the details. For simplicity, only exclusive locks are discussed; the logic is similar for shared locks.

12.8.1 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:



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.

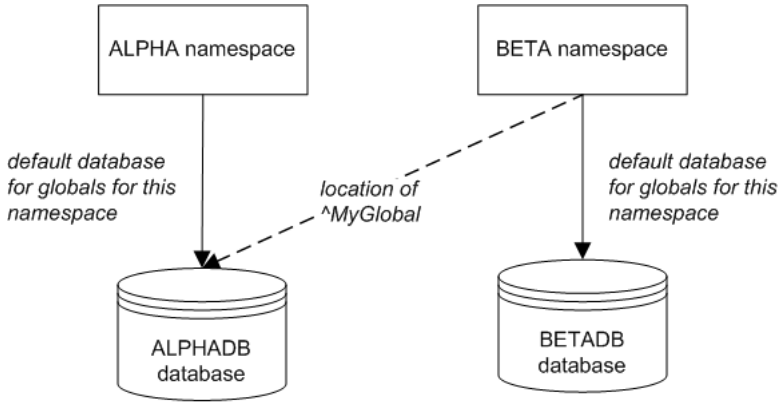
In this scenario, the lock table contains only the entry for the lock owned by Process A. If you examine the lock table, you will notice that it indicates the *database* to which this lock applies; see the **Directory** column. For example:

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\

12.8.2 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:



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\
   ```

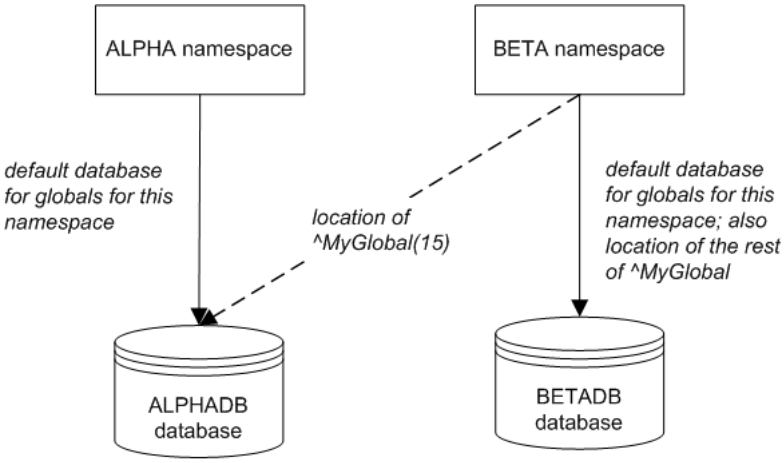
2. In namespace BETA, process B tries to acquire a lock with the name ^MyGlobal(15). This **LOCK** command does not return; the process is blocked until process A releases the lock.

12.8.3 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:



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.

12.8.4 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.

12.9 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 in the *Monitoring Guide*.

12.10 Practical Uses for Locks

This section presents the basic ways in which locks are used in practice.

12.10.1 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.

12.10.2 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")
```

12.11 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.

12.12 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

13

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.

13.1 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.

13.2 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.

13.3 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.

13.4 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.

13.5 See Also

- Locking and Concurrency Control
- Managing the Lock Table

14

Managing the Lock Table

This topic discusses tools for viewing and managing the lock table in InterSystems products. (Also see Monitoring Locks.)

14.1 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.

14.2 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 lock, escalating (`LOCK +^a(1)#"E"`)
Shared_e	A shared lock, escalating (`LOCK +^a(1)#"SE"`)
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.

14.3 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.

14.4 ^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.

14.5 See Also

- Locking and Concurrency Control
- Details of Lock Requests and Deadlocks

15

Transaction Processing

This page describes transaction processing in InterSystems IRIS® data platform. A transaction is a logical unit of work that combines multiple atomic operations in a specific sequence. An atomic operation is one that is always fully executed in any circumstance, including error conditions; such an operation typically corresponds to a single command.

15.1 About Transactions in InterSystems IRIS

In InterSystems IRS, atomic operations include a single SQL INSERT, UPDATE, or DELETE statement, or a single global **SET** or **KILL**.

However, an application typically needs to combine multiple atomic operations in order to accomplish a task. For example, when transferring money from one account to another, a bank may need to subtract an amount from a field in one table and add the same amount to a field in another table. By specifying that both updates form a single transaction, you ensure that either both operations are performed or neither is performed, which means that one cannot be executed without the other.

In such cases, you use transaction processing commands to define the sequence of operations that forms a complete transaction. One command marks the beginning of the transaction; after a sequence of possibly many commands, another command marks the end of the transaction.

Under normal circumstances, the transaction executes in its entirety. If a program error or system malfunction leads to an incomplete transaction, then the part of the transaction that was completed is rolled back.

Application developers should handle transaction rollback within their applications. InterSystems IRIS also handles transaction rollback automatically in the event of a system failure and at various junctures, such as recovery and during **HALT** or **ResJob**.

InterSystems IRIS records rollbacks in the messages.log file if the *LogRollback* configuration option is set. You can use the Management Portal, **System Operation**, **System Logs**, **Messages Log** option to view messages.log.

15.2 Managing Transactions Within Applications

In InterSystems IRIS, you define transactions within either SQL or ObjectScript, depending on your use case.

Important: These techniques are not fully interchangeable; you should manage any given transaction entirely within ObjectScript or entirely within SQL. For example, if you start a transaction in ObjectScript, then use ObjectScript commands to handle the rest of that transaction, matching TSTART to TCOMMIT, and so on. If you use the SQL commands, then you should use the SQL commands and match START TRANSACTION to COMMIT, and so on. You can, however, have these pieces of code call each other, and these pieces of code can be nested.

15.2.1 Transaction Commands

InterSystems IRIS supports the ANSI SQL operations **COMMIT WORK** and **ROLLBACK WORK** (in InterSystems SQL the keyword WORK is optional). It also supports the InterSystems SQL extensions **SET TRANSACTION**, **START TRANSACTION**, **SAVEPOINT**, and **%INTRANS**. In addition, InterSystems IRIS implements some of the transaction commands that are part of the M Type A standard.

These SQL and ObjectScript commands are summarized in the following table.

Table 15–1: Transaction Commands

SQL Command	ObjectScript Command	Definition
SET TRANSACTION		Set transaction parameters without starting a transaction.
START TRANSACTION	**TSTART**	Marks the beginning of a transaction.
%INTRANS	**$TLEVEL**	Detects whether a transaction is currently in progress: • <0 used by %INTRANS to mean in a transaction, but journaling disabled. Not used by **$TLEVEL**. • 0 means not in a transaction. • >0 means in a transaction.
SAVEPOINT		Mark a point within a transaction. Can be used for partial rollback to a savepoint.
COMMIT	**TCOMMIT**	Signals a successful end of transaction.
ROLLBACK	**TROLLBACK**	Signals an unsuccessful end of transaction; all the database updates performed since the beginning of transaction should be rolled back or undone.

These ObjectScript and SQL commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

15.2.2 Using LOCK in Transactions

Whenever you access a global which might be accessed by more than one process, you need to protect the integrity of the database by using the LOCK command on that global. You issue a lock corresponding to the global variable, change the

value of the global, then unlock the lock. The **LOCK** command is used to both lock and unlock a specified lock. Other processes wishing to change the value of the global request a lock which waits until the first process releases the lock.

There are three important considerations when using locks in transactions:

- Lock/unlock operations do not roll back.
- Within a transaction, when you unlock a lock held by the process, one of two things may occur:
  - The lock is immediately unlocked. The lock can be immediately acquired by another process.
  - The lock is placed in a delock state. The lock is unlocked, but cannot be acquired by another process until the end of the current transaction.

  If the lock is in a delock state, InterSystems IRIS defers the unlock until the transaction is committed or rolled back. Within the transaction, the lock appears to be unlocked, permitting a subsequent lock of the same value. Outside of the transaction, however, the lock remains locked. For further details, refer to Lock Management.
- Lock operations that time out set $TEST. A value set in **$TEST** during a transaction does not roll back.

15.2.3 Using $INCREMENT and $SEQUENCE in Transactions

A call to the $INCREMENT or $SEQUENCE function is not considered part of a transaction. It is not rolled back as part of transaction rollback. These functions can be used to get an index value without using the **LOCK** command. This is advantageous for transactions where you may not want to lock the counter global for the duration of the transaction.

$INCREMENT allocates individual integer values in the order that increment requests are received from one or more processes. **$SEQUENCE** provides a fast way for multiple processes to obtain unique (non-duplicate) integers for the same global variable by allocating a sequence (range) of integer values to each incrementing process.

Note: **$INCREMENT** may be incremented by one process within a transaction and, while that transaction is still processing, be incremented by another process in a parallel transaction. If the first transaction rolls back, there may be a "skipped" increment, "wasting" a number.

15.2.4 Transaction Rollback within an Application

If you encounter an error during a transaction, you can roll it back in three ways:

- Issue the SQL rollback command, **ROLLBACK WORK**
- Issue the ObjectScript rollback command, **TROLLBACK**
- Make a call to **%ETN**

Note: When you roll back a transaction, the IDKey for any default class is not decremented. Rather, the value of the IDKey is automatically modified by the **$INCREMENT** function.

15.2.4.1 Issue an SQL or ObjectScript Rollback Command

Application developers can use two types of rollback commands to designate the unsuccessful end of a transaction and automatically roll back incomplete transactions:

- Use **##sql(ROLLBACK WORK)**, in the macro source routine.
- Use the ObjectScript **TROLLBACK** command, in macro or intermediate source code.

The rollback command must cooperate with an error trap, as in the following example:

ObjectScript

```
ROU         ##sql(START TRANSACTION) set $ZT="ERROR"
             SET ^ZGLO(1)=100
             SET ^ZGLO=error
             SET ^ZGLO(1,1)=200
             ##sql(COMMIT WORK) Write !,"Transaction Committed" Quit
ERROR         ##sql(ROLLBACK WORK)
             Write !,"Transaction failed." Quit
```

In the example code, *$ZT* is set to run the subroutine ERROR if a program error occurs before the transaction is committed. Line **ROU** begins the transaction and sets the error trap. Lines ROU+1 and ROU+3 set the nodes of the global *^ZGLO*. However, if the variable error is undefined, **ROU+2** causes a program error and line **ROU+3** does not execute. Program execution goes to the subroutine **ERROR** and the set of *^ZGLO(1)* is undone. If line **ROU+2** were deleted, *^ZGLO* would have its value set both times, the transaction would be committed, and the message "Transaction committed" would be written.

15.2.4.2 Make a Call To %ETN

If you have not handled transaction rollback with a rollback command, the error trap utility **%ETN** detects incomplete transactions and prompts the user to either commit or rollback the transaction. You should handle rollback within your application, since committing an incomplete transaction usually leads to degradation of logical database integrity.

If you run **%ETN** after an error when a transaction is in progress, the following rollback prompt is displayed:

```
You have an open transaction.
Do you want to perform a Commit or Rollback?
Rollback =>
```

If there is no response within a 10-second timeout period, the system defaults to rollback. In a jobbed job or an application mode job, the transaction is rolled back with no message.

%ETN itself does not do anything to trigger transaction rollback, but it typically ends by halting out of InterSystems IRIS. Transaction rollback occurs when you halt out of ObjectScript and the system runs **%HALT** to perform InterSystems IRIS process cleanup. There is an entry point into **%ETN**, called **BACK^%ETN**, which ends with a quit, rather than a halt. If a routine calls **BACK^%ETN**, rather than **^%ETN** or **FORE^%ETN**, it will not perform transaction rollback as part of the error handling process.

15.2.5 Examples of Transaction Processing Within Applications

The following example shows how transactions are handled in macro source routines. It performs database modifications with SQL code. The SQL statements transfer funds from one account to another:

ObjectScript

```
Transfer(from,to,amount)   // Transfer funds from one account to another
{
   TSTART
    &SQL(UPDATE A.Account
         SET A.Account.Balance = A.Account.Balance - :amount
         WHERE A.Account.AccountNum = :from)
    If SQLCODE TRollBack  Quit "Cannot withdraw, SQLCODE = "_SQLCODE
    &SQL(UPDATE A.Account
         SET A.Account.Balance = A.Account.Balance + :amount
         WHERE A.Account.AccountNum = :to)
    If SQLCODE TROLLBACK  QUIT "Cannot deposit, SQLCODE = "_SQLCODE
    TCOMMIT
    QUIT "Transfer succeeded"
}
```

15.3 Automatic Transaction Rollback

Transaction rollback occurs automatically during:

- InterSystems IRIS startup, if recovery is needed. When you start InterSystems IRIS and it determines that recovery is needed, any transaction on the computer that was incomplete will be rolled back.
- Process termination using the HALT command (for the current process) or the ^RESJOB utility (for other processes). Halting a background job (non-interactive process) automatically rolls back the changes made in the current transaction-in-progress. Halting an interactive process prompts you whether to commit or roll back the changes made in the current transaction-in-progress. If you issue a **^RESJOB** on a programmer mode user process, the system displays a message to the user, asking whether they want the current transaction committed or rolled back.

In addition, system managers can roll back incomplete transactions in cluster-specific databases by running the **^JOURNAL** utility. When you select the `Restore Globals From Journal` option from the **^JOURNAL** utility main menu, the journal file is restored and all incomplete transactions are rolled back.

15.4 System-Wide Issues with Transaction Processing

This section describes various system-wide issues related to transaction processing. For more information on issues related to backups, see Backup and Restore; for more information on issues related to ECP, see ECP Recovery Process, Guarantees, and Limitations.

15.4.1 Backups and Journaling with Transaction Processing

Consider the following backup and journaling procedures when you implement transaction processing.

Each instance of InterSystems IRIS keeps a journal. The journal is a set of files that keeps a time-sequenced log of changes that have been made to the database since the last backup. InterSystems IRIS transaction processing works with journaling to maintain the logical integrity of data.

The journal contains **SET** and **KILL** operations for globals in transactions regardless of the journal setting of the databases in which the affected globals reside, as well as all **SET** and **KILL** operations for globals in databases whose **Global Journal State** you set to "Yes."

Backups can be performed during transaction processing; however, the resulting backup file may contain partial transactions. In the event of a disaster that requires restoring from a backup, first restore the backup file, and then apply journal files to the restored copy of the database. Applying journal files restores all journaled updates from the time of the backup, up to the time of the disaster. Applying journals is necessary to restore the transactional integrity of your database by completing partial transactions and rolling back uncommitted transactions, since the databases may have contained partial transactions at the time of the backup. For detailed information, see:

- Journaling
- Importance of Journals

15.4.2 Asynchronous Error Notifications

You can specify whether a job can be interrupted by asynchronous errors using the **AsynchError()** method of the %SYSTEM.Process class:

- **%SYSTEM.Process.AsynchError(1)** enables the reception of asynchronous errors.
- **%SYSTEM.Process.AsynchError(0)** disables the reception of asynchronous errors.

The *AsynchError* property of the Config.Miscellaneous class sets a system-wide default for new processes for whether processes are willing to be interrupted by asynchronous errors. It defaults to 1, meaning "YES."

If multiple asynchronous errors are detected for a particular job, the system triggers at least one such error. However, there is no guarantee which error will be triggered.

The asynchronous errors currently implemented include:

- <LOCKLOST> — Some locks once owned by this job have been reset.
- <DATALOST> — Some data modifications performed by this job have received an error from the server.
- <TRANLOST> — A distributed transaction initiated by this job has been asynchronously rolled back by the server.

Even if you disable a job receiving asynchronous errors, the next time the job performs a **ZSync** command, the asynchronous error is triggered.

At each **TStart**, **TCommit**, or **LOCK** operation, and at each network global reference, InterSystems IRIS checks for pending asynchronous errors. Since **SET** and **KILL** operations across the network are asynchronous, an arbitrary number of other instructions may interpose between when the **SET** is generated and when the asynchronous error is reported.

15.5 Suspending All Current Transactions

You can use the **TransactionsSuspended()** method of the %SYSTEM.Process class to suspend all current transactions for the current process. This is a boolean method: 1 = suspend all current transactions, 0 = resume all current transactions. The default is 0.

While transactions are suspended changes are not logged to the transaction log and therefore cannot be rolled back. Change made in the current transaction before or after an interval when transactions were suspended can be rolled back.

If you change a global variable in a transaction, and then change it again while that transaction is suspended may result in an error when rollback is attempted.

Invoking **TransactionsSuspended()** without specifying a boolean parameter returns the current boolean setting without changing that setting.

16

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.

16.1 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.

16.2 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.

16.3 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.

16.4 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.

16.5 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
```

16.6 %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.

16.7 See Also

For more information, see the class reference for the %SYSTEM.Status class and the %Status (%Library.Status) class.

17

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.

17.1 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*.

17.2 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.

17.3 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.

17.4 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

17.5 Other Considerations with TRY-CATCH

The following describe conditions that may arise when using a **TRY-CATCH** block.

17.5.1 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.

17.5.2 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.

17.5.3 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.

18

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.

18.1 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.

18.2 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.

18.3 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.

18.4 See Also

- **%SYS.ProcessQuery.ExamStackByPid()** method, which provides details on the ^*mtemp* global used by **^%ERN**

19

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.

19.1 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.

19.1.1 Restricted Commands and Functions

This section lists the restricted activities within the secure debug shell:

- Restricted ObjectScript Commands
- Restricted ObjectScript Functions
- Restricted Object Constructions

19.1.1.1 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**

19.1.1.2 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

19.1.1.3 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.

19.2 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.

19.2.1 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.

19.2.2 Establishing Breakpoints and Watchpoints

You use the ZBREAK command to establish breakpoints and watchpoints.

19.2.2.1 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.

19.2.2.2 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.

19.2.2.3 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.

19.2.2.4 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.

19.2.2.5 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.

19.2.2.6 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
```

19.2.3 Disabling Breakpoints and Watchpoints

You can disable either:

- Specific breakpoints and watchpoints
- All breakpoints or watchpoints

19.2.3.1 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
```

19.2.3.2 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.

19.2.4 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.

19.2.5 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.

19.2.6 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"
```

19.2.7 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.

19.2.7.1 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.

19.2.8 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**

19.2.9 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

19.2.9.1 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

19.2.9.2 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.

19.2.10 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.

19.2.11 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.

19.2.12 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
```

19.3 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.)

19.3.1 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.

19.3.2 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.

19.3.2.1 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-"
```

19.3.3 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 19–1: 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>
```

19.3.4 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.

19.3.5 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>
```

19.3.5.1 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>
```

19.3.6 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.

19.3.7 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>
```

19.3.8 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.

19.4 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.

19.4.1 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.

19.4.2 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.

19.4.2.1 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.

19.4.2.2 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>
```

19.4.2.3 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.

19.4.2.4 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

19.4.3 Understanding the Stack Display

Each item on the stack is called a *frame*. The following table describes the information provided for each frame.

Table 19–2: %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 19–3: 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.			

19.4.3.1 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.

19.4.3.2 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.

19.4.3.3 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:
```

19.4.3.4 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

19.4.3.5 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.

19.5 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

19.5.1 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.

19.5.2 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.

19.5.2.1 %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")
```

19.5.2.2 %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.")

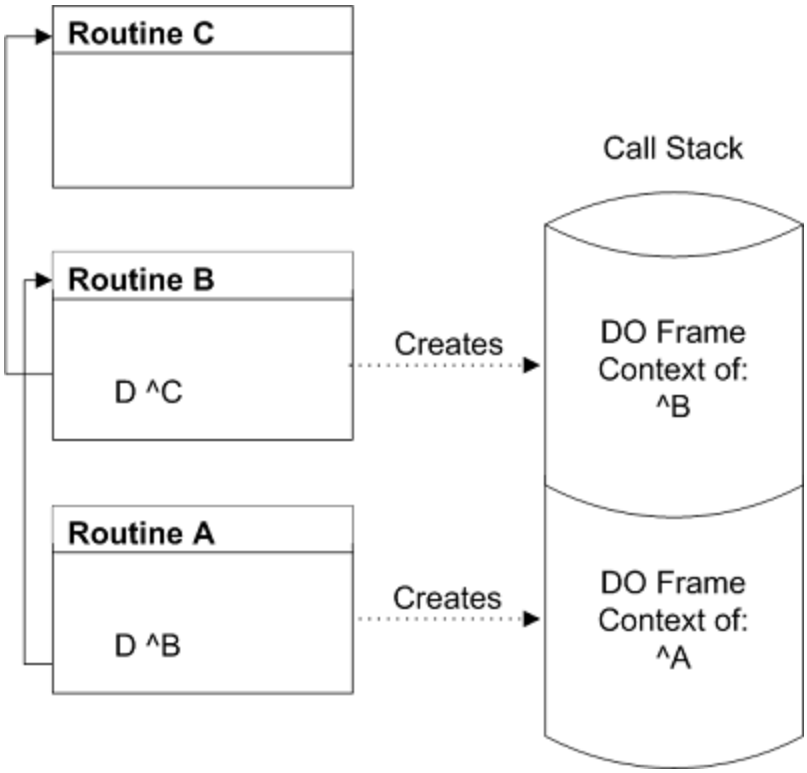
- An **XECUTE** command argument causes ObjectScript code to execute. (This kind of frame is also known as a "**XECUTE** frame.")
- A user-defined function is executed.

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.

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.

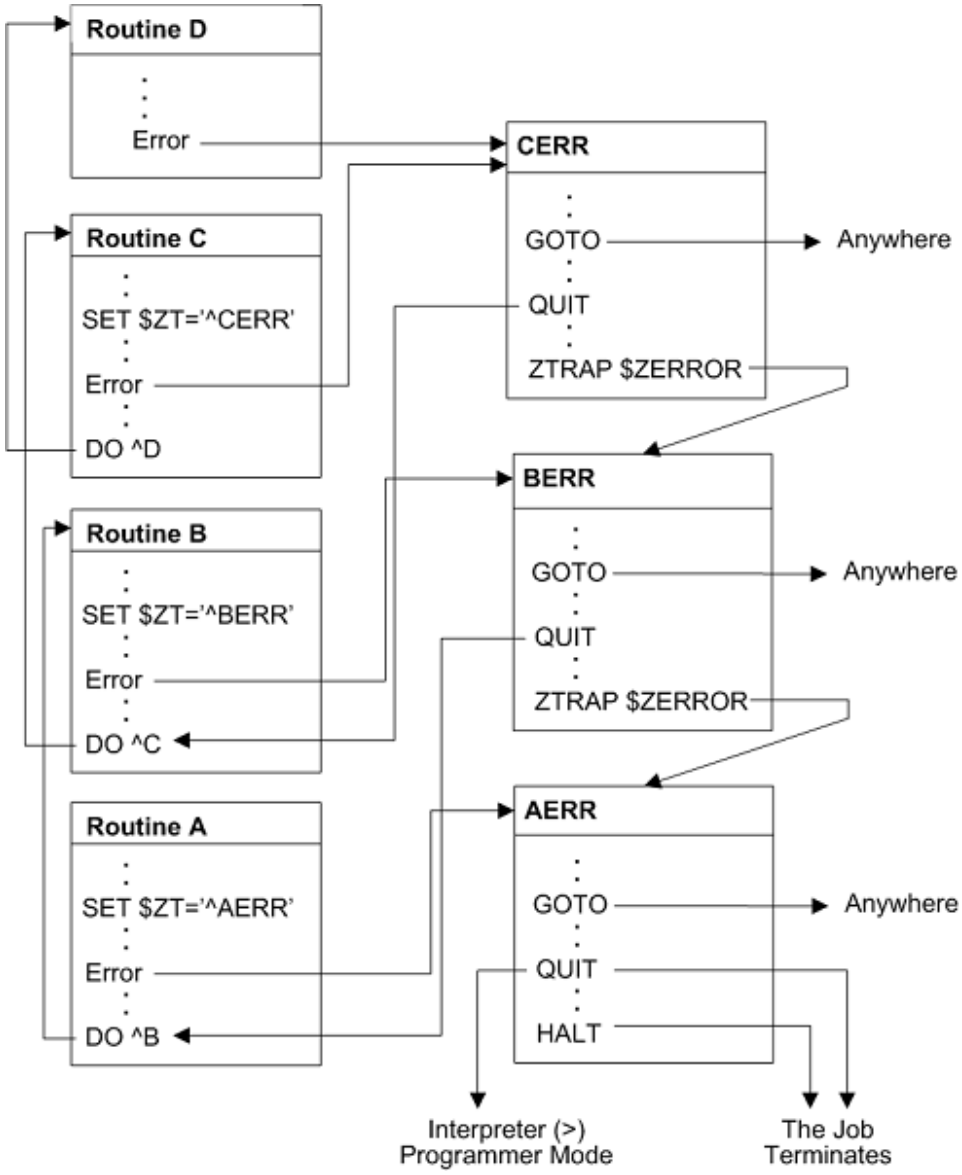
When an error handler has handled an error, the **$ZERROR** special variable is set to a value. This value is not necessarily cleared when the error handler completes. Some routines reset **$ZERROR** to the null string. The **$ZERROR** value is overwritten when the next error occurs that invokes an error handler. For this reason, the **$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.

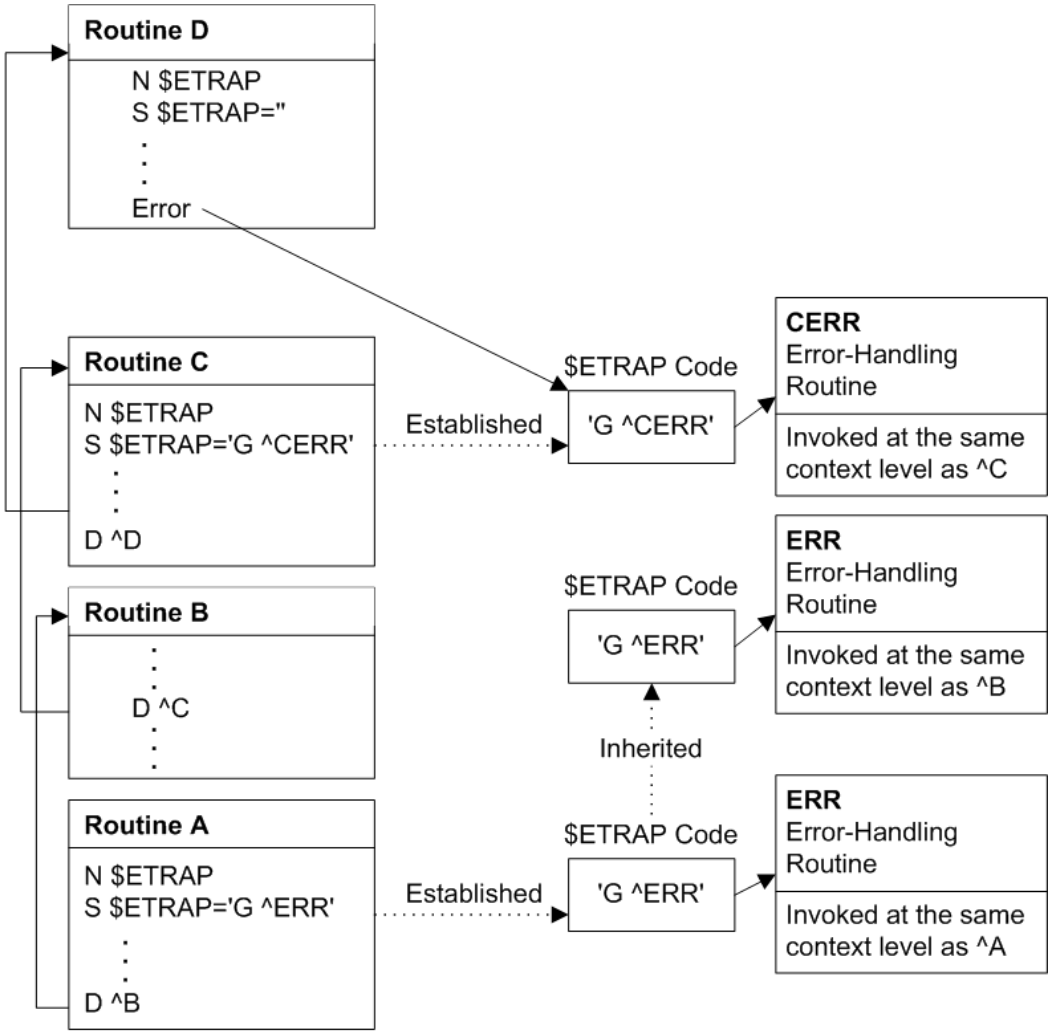
Keep in mind that creating a new copy of **$ETRAP** does not clear **$ETRAP**. The value of **$ETRAP** remains unchanged by the **NEW** command.

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.