Using ObjectScript

Version 2019.4
2020-01-28

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

Using ObjectScript
InterSystems IRIS Data Platform Version 2019.4 2020-01-28

For Support questions about any InterSystems products, contact:

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

Table of Contents

List of Figures

List of Tables

About This Book

This book describes the various elements of the ObjectScript programming language.

Its topics are:

- Introducing ObjectScript
- Syntax Rules
- Data Types and Values
- Variables
- Operators and Expressions
- Regular Expressions
- Commands
- User-defined Code
- ObjectScript Macros and the Macro Preprocessor
- Embedded Code
- Multidimensional Arrays
- String Operations
- Lock Management
- Transaction Processing
- Error Processing
- Command-line Routine Debugging

For a detailed outline, see the Table of Contents.

Other, related documents in the documentation set are:

- *The ObjectScript Language Reference*
- *Orientation Guide for Server-Side Programming*
- *Defining and Using Classes*
- *Using InterSystems SQL*
- *Using Globals*

1

Introducing ObjectScript

ObjectScript is an object programming language designed for rapidly developing complex business applications on InterSystems IRIS® data platform. It is well-suited for a variety of applications including:

- Business logic
- Application integration
- Data processing

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:

- Interactively from the command line of the Terminal.
- As the implementation language for methods of InterSystems IRIS object classes.
- To create ObjectScript routines: individual programs contained and executed within InterSystems IRIS.
- As the implementation language for Stored Procedures and Triggers within InterSystems SQL.

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.

1.1 Features

Some of the key features of ObjectScript include:

- Powerful built-in functions for working with strings.
- Native support for objects including methods, properties, and polymorphism.
- A wide variety of commands for directing control flow within an application.
- 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

The following is an introduction to the major elements of ObjectScript.

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:

```
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:

```
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 refer to "Case Sensitivity" in the "Syntax Rules" chapter of this document.

Whitespace can be inserted or omitted almost anywhere in ObjectScript. However, one use of whitespace is significant; a command cannot start on the first character position on a line, nor can a command continue on the first character position of a line. Thus, all command lines must be indented. Comments must also be indented. However, a label must begin on the first character position of a line, and some other syntax, such as macro preprocessor statements, may begin on the first character position of a line. For details refer to "Whitespace" in the "Syntax Rules" chapter of this document.

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

1.3 Invoking Commands and Functions

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

```
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:

```
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). Whitespace is discussed in the Syntax chapter of this document.

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:

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

The short form of a command name is simply a device for developers who do not like to type long command names. It is exactly equivalent to the long form. This document uses the long form of command names. For a complete list, see Abbreviations Used in ObjectScript in the *ObjectScript Reference*.

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

1.3.2 Functions

A *function* is a routine 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 in the *ObjectScript Reference*.

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 (such as moving a pointer), 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. A function does not modify the value of a variable supplied as an input parameter, 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 (optionally) sets its third parameter 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:

```
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, but they do not have to be typed. Variables in ObjectScript are untyped; that is, they do not have an assigned data type and can take any data value. (For compatibility, the $DOUBLE function can be used to convert untyped floating point numbers to a specific numeric data type format.)

ObjectScript supports several kinds of variables:

- 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. A local variable is accessible from any namespace.
- Process-private globals — A variable that is accessible only by the InterSystems IRIS process and is deleted when the process ends. A process-private global is accessible from any namespace. 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. Globals are specific to individual namespaces.
- 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 are defined; InterSystems IRIS sets all special variables to an initial value (sometimes a null string value). 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 much greater detail in the Variables chapter of this document.

1.3.5 Operators

ObjectScript defines a number of built-in operators. These include arithmetic operators, such as addition (“+”) and multiplication (“*”), logical operators, and pattern match operators. For details, refer to the Operators chapter of this document.

2

Syntax Rules

This chapter describes the basic rules of ObjectScript syntax on InterSystems IRIS® data platform. Topics include:

2.1 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.

2.1.1 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: InterSystems IRIS SQL identifiers, in contrast, are *not* case-sensitive.

2.1.2 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.

2.1.3 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.

2.1.4 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 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.

2.2 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. All characters are encoded using 16 bits (two bytes). However, when saving a string to disk in a global, if all the characters in the string have numerical values of 255 or lower (ASCII values), InterSystems IRIS stores the string using one byte per character. You can work with characters that are not in the Unicode Basic Multilingual Plane by using $WCHAR, $WISWIDE, and related functions.

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.

2.2.1 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 ASCII 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.

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).

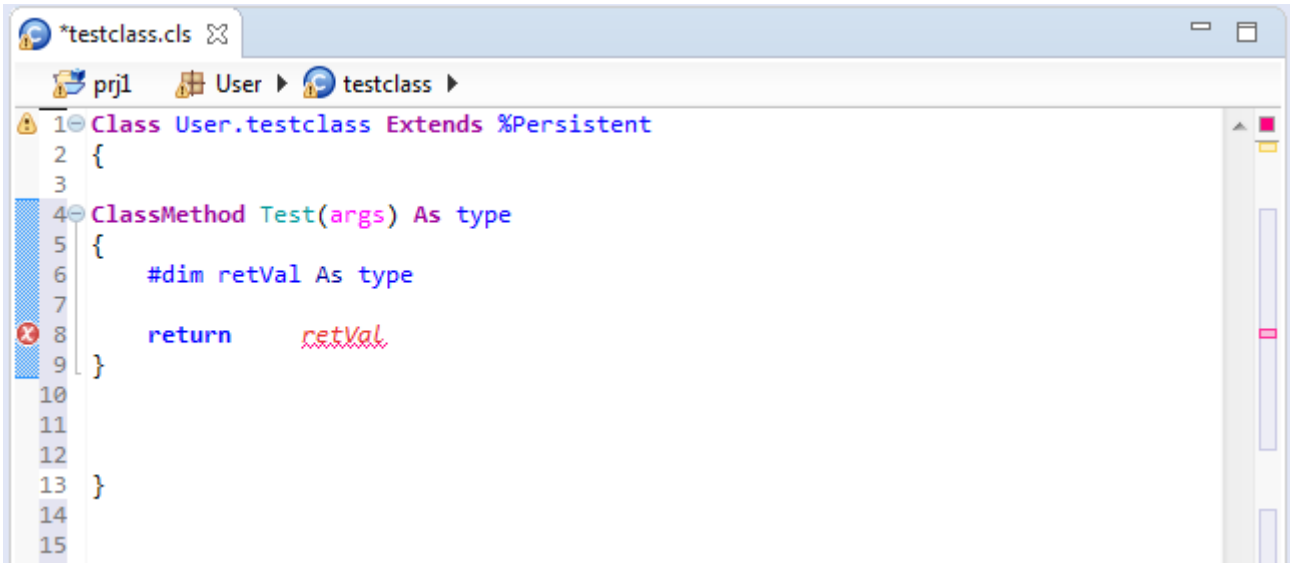
2.3 Whitespace

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.

Atelier provides built-in syntax checking, so that it will mark any illegal use of whitespace, such as the following insertion of multiple spaces before a command's first argument:

Figure 2–1: Atelier Syntax Checking



2.4 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

2.4.1 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:

```
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"
```

2.4.2 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.

2.4.3 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 either "Creating Class Documentation" in the chapter Defining and Compiling Classes in *Defining and Using Classes* or the %CSP.Documatic entry in the *InterSystems Class Reference*.

2.5 Literals

A literal is a constant value consisting of a series of characters that represents a particular string or numeric value, such as “Hello” and “5” below:

```
SET x = "Hello"
SET y = 5
```

ObjectScript recognizes two types of literals:

- String literals
- Numeric literals

2.5.1 String Literals

A string literal is a set of zero or more characters delimited by quotation marks (unlike string literals, numeric literals do not need 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. The length of a string literal is measured as the number of characters in the string, not the number of bytes. A string can contain just 8-bit ASCII characters. Or a string can also contain 16-bit Unicode characters (known as wide characters). If you include a single wide character in a string, InterSystems IRIS will represent all characters in the string as 16-bit characters. Because InterSystems IRIS almost always treats a string as characters, not bytes, this use of wide characters is usually invisible to the user. (One exception is the **ZZDUMP** command, as shown below.)

The maximum string size is 3,641,144 characters.

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

```
  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:

```
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:

```
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:

```
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:

```
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, refer to Strings in the "Data Types and Values" chapter of this manual.

2.5.2 Numeric Literals

Numeric literals are values that ObjectScript evaluates as numbers. They do not require delimiters. InterSystems IRIS converts numeric literals to canonical form (their simplest numeric form):

```
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:

```
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 in the "Data Types and Values" chapter of this manual.

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 in the "Data Types and Values" chapter.

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).

2.6 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. Refer to Extending Languages with ^%ZLANG Routines in *Specialized System Tools and Utilities*.

Naming conventions for local variables, process-private globals, and globals are provided in the Variables chapter of this document.

2.6.1 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" in the *Orientation Guide for Server-Side Programming*.
- 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.

2.7 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:

```
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.

2.7.1 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.

2.7.2 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":

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",!
           QUIT }
  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:

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",!
           QUIT }
  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:

```
  SET x = $RANDOM(2)
  IF x=0 {DO label0
           WRITE "Finished Routine0",!
           QUIT }
  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"
```

2.8 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 IRIS instance. To return the full pathname of the current namespace, you can invoke the **NormalizeDirectory()** method, as shown in the following example:

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

For information on using namespaces, see Namespaces and Databases in the *Orientation Guide for Server-Side Programming*. For information on creating namespaces, see Configuring Namespaces in the *System Administration Guide*.

2.8.1 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, refer to Global Variables section of the “Variables” chapter of this manual.

- 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.

2.9 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, such as:

- Strings
- Numbers
- Objects
- Persistent Multidimensional Arrays (Globals)
- Undefined Values
- Boolean Values
- Dates

3.1 Strings

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

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

Topics about strings include:

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

3.1.1 Maximum String Length

InterSystems IRIS® data platform supports a maximum string length of 3,641,144 characters. Attempting to exceed this maximum string length results in a <MAXSTRING> error.

When a process uses a string, the memory for the string comes from the operating system's malloc() buffer, not from the partition memory space for the process. Thus the memory allocated for actual string values is not subject to the limit set by the maximum memory per process (Maximum per Process Memory (KB)) parameter and does not affect the $STORAGE value for the process.

3.1.2 Null String / $CHAR(0)

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

```
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:

```
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. Refer to NULL and the Empty String in the "Language Elements" chapter of *Using InterSystems SQL*.

3.1.3 Escaping Quotation Marks

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

```
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.

3.1.4 Concatenating Strings

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

```
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:

```
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, 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".

3.1.5 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:

```
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.

3.1.5.1 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.

3.1.6 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:

```
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
- Fractional 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.
- 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:

```
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:

  ```
  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 the integer zero from fractions; for example `0.66` becomes `.66`.

   - To append an integer zero to a canonical fraction use the $FNUMBER or $JUSTIFY function. `.66` becomes `0.66`.
   - To remove an integer zero from a non-canonical fraction use the Unary Plus operator. For example `+$PIECE($ZTMESTAMP,",",2).0.66` 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, refer to 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, refer to String-to-Number Conversion in the "Operators and Expressions" chapter of this book.

- 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:

```
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:

```
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:

```
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.

3.2.4 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:

```
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 Fractional Numbers

InterSystems IRIS supports two different numeric types that can be used to represent fractional 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 the highest level of precision — 18 decimal digits. 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 binary floating-point.

- Binary floating-point: The IEEE double-precision binary floating point standard 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. Double-precision binary floating point has a precision of 53 binary bits, which corresponds to 15.95 decimal digits of precision. 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 standard InterSystems IRIS 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 standard InterSystems IRIS fractional numbers. IEEE floating point numbers are able to represent larger and smaller numbers than standard InterSystems IRIS numbers, and support the special values INF (infinity) and NAN (not a number). For further details, refer to the $DOUBLE function.

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:

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

where

-	*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:

```
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:

```
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:

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

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

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

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

```
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
}
```

```
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:

```
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 the appendix "Numeric Computing in InterSystems Applications" in the *Orientation Guide for Server-Side Programming*.

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:

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

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

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

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

```
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:

```
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:

```
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 the "Multidimensional Arrays" chapter in this document and the Using Globals document.

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:

```
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:

```
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:

```
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:

```
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, refer to String-to-Number Conversion in the “Operators and Expressions” chapter of this book.

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:

Date Formats

```
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 used. 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.

4.1 Categories of Variables

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

- Local variables
- Process-private global variables (PPGs)
- Global variables (also known as *globals*)
- i%property instance variables
- Special variables (also known as *system variables*)

Each of these is used for a specific purpose and may have different scoping rules.

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. For further details on these topics refer to Variable Declaration and Variable Typing and Conversion.

4.1.1 Subscripted Variables

Local variables, process-private variables, and global variables can all take subscripts. The subscript conventions for all types of variables are similar:

- 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.
- A numeric subscript is converted to canonical form. Thus, ^a(7), ^a(007), ^a(7.000), and ^a(7.) are all the same subscript. A string subscript is not converted to canonical form. Thus ^a(“7”), ^a(“007”), ^a(“7.000”), and ^a(“7.”) are different subscripts. The string subscript ^a(“7”) is the same as the numeric subscript ^a(7).
- The maximum length of a subscript is 511 encoded bytes (the corresponding number of characters depends on the characters in the subscript and the current locale). Exceeding the maximum subscript length results in a <SUBSCRIPT> error. However, the longest permitted integer is 309 digits; exceeding this limit results in a <MAXNUMBER> error. Therefore, a numeric subscript longer than 309 characters must be specified as a string.
- The maximum number of subscript levels for a local variable is 255. The maximum number of subscript levels for a global or process-private global is 253. Exceeding the maximum number of subscript levels results in a <SYNTAX> error.

Actual maximums depend on several factors. For further details on global maximums, refer to Global Structure chapter in *Using Globals*.

Lock name subscripts follow the same conventions as variable subscripts.

4.1.1.1 Array Variables

An array variable is simply a variable with one or more subscript levels. Subscripts are enclosed in parentheses. Subscript levels are separated by commas. Any variable (with the exception of special variables) can be used as an array, as shown in the following example:

```
SET a(1) = "A local variable array"
SET a(1,1,1) = "Another local variable array"
SET ^||a(1) = "A process-private global array"
SET ^a(1) = "A global array"
SET obj.a(1) = "A multidimensional array property"
```

For local variables, the maximum number of subscript levels is 255. For global variables, the maximum number of subscript levels depends on the length of the subscript names. For subscript conventions and limits, refer to the Global Structure chapter in Using Globals.

4.1.2 Object Properties

An object property is a value associated with, and stored within, a specific instance of an object. Strictly speaking, an object property is not a variable, but syntactically you can use an object property in exactly the same way as any other local variable.

There are several types of property references: `oref.property`, `..property`, and `i%property`. The object reference (`oref`) must be a local variable, not an expression. The `property` reference can reference a single-value property or a subscripted multidimensional property. There can be chained property references like `oref.prop1.prop2`.

The following example shows a property reference used as a variable:

```
// Create an Address object
SET address = ##class(Sample.Address).%New()
// Use the properties of the object
SET address.City = "Boston"
WRITE "City: ",address.City,!
```

There are some limitations for the use of property references as variables. A property of any type cannot be modified using SET $BIT() or SET $LISTBUILD(); a non-multidimensional property cannot be modified using SET $EXTRACT(), SET $LIST(), or SET $PIECE(). $DATA(), $GET(), and $INCREMENT() can only take a property if the property is multidi-

mensional. Properties cannot be used with the MERGE command. These operations can be done within an object method using the instance variable syntax `i%PropertyName`.

4.2 Local Variables

A local variable is a variable that is stored within the current InterSystems IRIS process. It is accessible only to the process that created it. It is mapped to be accessible from all namespaces. When a process ends, all of the processes' local variables are deleted.

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

4.2.1 Naming Conventions

Define a local variable using the following naming conventions:

- A local variable name must be a valid identifier. Its 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. Only variables that begin with "%Z" or "%z" are available for application code; all other percent variables are reserved for system use according to the rules described in "Rules and Guidelines for Identifiers" in the *Orientation Guide for Server-Side Programming*. The percent (%) character can only be used as the first character of a local variable name. The other characters of a local variable name may be letters or numbers.
- Any word can be used as a variable name. However, it is *strongly* recommended that a variable name not be the name of an ObjectScript command or an SQL reserved word.
- Local variable names are case-sensitive. For example: MYVAR, MyVar, and myvar are three different local variables.
- Local variable names may include letter characters above ASCII 255 (Unicode letters).
- 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 name as a local variable. For example: myvar, ^||myvar, and ^myvar are three different variables.
- Local variable names are limited to 31 characters. You may specify a name longer than 31 characters, but only the first 31 characters are used. Therefore, a local variable name must be unique within its first 31 characters.
- Local variables can take subscripts. By using subscripts, you can define a local variable as an array of values. A subscript can be a number or a character string; it can include Unicode characters.

Note: The %IS utility sets several local variables with all-uppercase names. These variable names should be avoided in situations where %IS is invoked. For further details, see I/O Devices and Commands in the *I/O Device Guide*.

4.2.1.1 Invalid Names

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

A valid variable name prefaced by a caret (^) is a global variable, not a local variable.

A valid variable name prefaced by an at sign (@) is an indirection operator followed by a local variable name.

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" in the *Orientation Guide for Server-Side Programming*. You can use the NEW command on a % variable in a procedure block.

  These mechanisms are described in greater detail in the Procedure Variables section of the "Callable User-defined Code Modules" chapter.
- 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 $QSUBSCRIPT function to return the components (name and subscripts) of a specified local variable, or the $QLENGTH function to return the number of subscript layers. You can use the **KILL** command to delete local variables.

4.2.3 Object Values

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

```
SET person = ##class(Sample.Person).%New()
WRITE person,!
```

Note: The value of *person* is that of an object reference (OREF) converted into a string. This string or its value cannot be used to load an object from the database.

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

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

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

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

WRITE "Is string an object? ", $IsObject(str),!
WRITE "Is person an object? ", $IsObject(person),!
```

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

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

```
SET person = ##class(Sample.Person).%New() // one reference to Person
SET alias = person // two references

SET person = "" // 1 reference

SET alias = "" // no references left, object destroyed
```

4.3 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 cease 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 exists independent of namespace. It is mapped to be accessible from all namespaces. Thus it can be created, accessed, and deleted regardless of current namespace. This differs from both local variables and global variables, which are both namespace-specific.
- 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.

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

4.3.1 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. They allow you to specify a variable that determines whether to define *name* as a process-private global or a standard global. This is shown in the following example:

```
SET x=1       // toggle storage type
IF x=1 {
  SET a="^"   // for a process-private global
}
ELSE {
  SET a=""    // for a standard global
}
SET ^|a|name="a value"
```

Process-private globals use the following naming conventions:

- A process-private global name must be a valid identifier. Its first character (after the second vertical bar) must be either a letter or the percent (%) character. The percent (%) character can only be used as the first character of a process-private global name. Only percent variables that begin with "%Z" or "%z" are available for application code (such as `^||%zmyppg` or `^||%Z123`); all other percent variables are reserved for system use according to the rules described in "Rules and Guidelines for Identifiers" in the *Orientation Guide for Server-Side Programming*.

  The second and subsequent characters of a process-private global name may be letters, numbers, or the period character. A period cannot be used as the first or last character of the name.

- A process-private global name cannot 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.
- Process-private global names are case-sensitive.
- A process-private global name must be unique within its process.
- 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.
- Process-private globals can take subscripts. By using subscripts, you can define a process-private global as an array of values. A subscript can be a number or a character string; it can include Unicode characters. For subscript conventions, limitations on the number of subscript levels, and other information about use of subscripts, refer to Global Structure in *Using Globals*.

4.3.2 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 utility 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 specify 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:

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

^GETPPGINFO has the following syntax:

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

The *pdf* argument can be a process Id or the * wildcard. The *options* argument can be a string containing any combination of the following: b (return values in bytes), M*nn* (return only those process-private variables that use *nn* or more blocks); S (suppress screen display; used with *outfile*); T (display process totals only). The *outfile* argument 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 variables to an output file named `ppgout`. The S option suppresses screen display; the M500 option limits output to only process-private variables that use 500 or more blocks:

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

4.4 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.

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 to changes to a global until the transaction that made the changes has been committed. Refer to the "Transaction Processing" chapter for further details.

Within an ObjectScript program, you can use a global in the same way as any other variable. Syntactically, a global name is distinguished by a caret ("^") character followed by a letter or a "%" character:

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

The naming conventions for globals are as follows:

- A global consists of a global prefix and a global name. The global prefix is commonly a caret (^) character, specifying a global in the current namespace. A global prefix can also be an extended reference, such as `^|"samples"|`, specifying a global in another namespace.
- A global name must be a valid identifier. Its first character (after the prefix character(s)) must be either a letter or the percent (%) character. Only percent variables that begin with "%Z" or "%z" are available for application code (such as `^%zmyglobal` or `^%Z123`); these ^%Z and ^%z globals are written to the IRISSYS database, and are therefore preserved when you upgrade InterSystems IRIS. All other percent variables are reserved for system use according to the rules described in "Rules and Guidelines for Identifiers" in the *Orientation Guide for Server-Side Programming*.

  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.
- A global name cannot 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.

- Globals can take subscripts. By using subscripts, you can define a global as an array of values. A subscript can be a number or a character string; it can include Unicode characters. For subscript conventions, limitations on the number of subscript levels, and other information about use of subscripts, refer to Global Structure in *Using Globals*.
- Some global names are reserved for InterSystems use. For further details, refer to "Global Variable Names to Avoid" in the *Orientation Guide for Server-Side Programming*.

Optionally, a global may specify an extended reference that defines its namespace or directory using a pair of vertical bars or square brackets immediately after the caret characters (for example: `^|"samples"|myglobal` or `^|""|myglobal`). These extended global references should not be confused with process-private globals.

You can use the $ZREFERENCE special variable to determine the name of the most recently used global. You can use the $QSUBSCRIPT function to return the components of a specified global, or the $QLENGTH function to return the number of subscript layers.

For much more information on globals, refer to Using Globals.

4.5 Special Variables

ObjectScript includes a number of built-in 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 set of special variables is mapped to be accessible from 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:

```
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:

```
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 in the "Commands" chapter of *Using ObjectScript*.

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

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:

```
// 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 the "String-to-Number Conversion" section of the "Operators and Expressions" chapter. 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 Variable Declaration

Unlike other languages, you do not declare variables in ObjectScript. You can, however, use the #Dim preprocessor directive as an aid to documenting code.

When writing code in Studio, you can use the #Dim preprocessor directive. **#Dim** provides information on the intended type of a variable. Studio uses this information for code completion with the Studio Assist feature. This information can also be used for documentation or to provide information to others who may view the code. (For more information on Studio Assist, see the section "Editor Options" in the "Setting Studio Options" chapter of *Using Studio*.)

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

```
#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. Studio provides a menu from which you can select the *DataTypeName* value.

#Dim also allows you to specify an initial value for a variable, as in the following:

```
#Dim President As %String = "Obama"
```

5

Operators and Expressions

InterSystems IRIS® data platform 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 chapter describes expressions and the various ObjectScript operators. It contains the following topics:

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. Such expressions produce a result that is the effect of the operator on the operand(s). They are classified based on the types of operators they contain.

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

- 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).

5.1.1 Table of Operator Symbols

ObjectScript includes the following operators:

Table 5–1: ObjectScript Operators

Operator	Operation Performed
.	Object property or method access.
()	Array index or function call arguments.
+	Addition (Binary), Positive (Unary)
–	Subtraction (Binary), Negative (Unary)
*	Multiplication
/	Division
\	Integer division
**	Exponentiation
#	Modulus (remainder)
_	Concatenation
'	Logical complement (NOT)
=	Test for equality, Assignment
'=	Test for non-equality
>	Greater than
'> <=	Not greater than (less than or equal to)
<	Less than
'< >=	Not less than (greater than or equal to)
[	Contains
]	Follows
]]	Sorts After
& &&	Logical AND (&& is “short-circuit” AND)
! ||	Logical OR (|| is “short-circuit” OR)
@	Indirection

Operator	Operation Performed
?	Pattern Match

These are described in more detail in the following sections.

5.1.2 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.

```
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.

5.1.2.1 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.

```
WRITE -123 - 3,!       // returns -126
WRITE -123 + -3,!      // returns -126
WRITE -(123 - 3),!     // returns -120
```

5.1.2.2 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:

```
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:

```
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:

```
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:

```
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:

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

5.1.2.3 Functions and Precedence

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

```
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:

```
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).

5.1.3 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:

```
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:

```
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:

```
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:

```
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:

  ```
  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.

  ```
  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):

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

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

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

5.1.3.1 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 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:

```
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:

```
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:

```
LogExp
 IF $$One(),$$Two() {
    WRITE !,"Expression is TRUE."  }
 ELSE {
    WRITE !,"Expression is FALSE." }
One()
 WRITE !,"one"
 QUIT 0
Two()
 WRITE !,"two"
 QUIT 1
```

5.1.4 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:

```
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:

```
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 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"`.

5.2.1 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:

```
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:

```
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:

```
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:

```
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
```

5.2.2 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* 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

5.3 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 the section "String-to-Number Conversion."

5.3.1 Decimal and $DOUBLE Floating-Point Numbers

InterSystems IRIS supports two representations of decimal-point numbers: ObjectScript decimal floating-point and IEEE double-precision binary floating-point.

- InterSystems IRIS represents a numeric constant as an ObjectScript decimal floating-point value by default. This is referred to as a $DECIMAL number. A $DECIMAL number can exactly represent a fractional value, such as 2.2. You

use the $DECIMAL() function to explicitly convert an IEEE double-precision binary floating-point number to the corresponding ObjectScript decimal floating-point number.

- InterSystems IRIS also supports IEEE double-precision binary floating-point values. You use the $DOUBLE() function to explicitly convert a numeric constant to the corresponding IEEE double-precision binary floating-point value (referred to as a $DOUBLE number). A $DOUBLE number can only approximately represent a fractional value, such as 2.2. $DOUBLE representation is usually preferred when doing high-speed scientific calculations because most computers include high-speed hardware for binary floating-point arithmetic.

ObjectScript automatically converts a numeric 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. For example, `2.2 + $DOUBLE(.1)` is the same as `$DOUBLE(2.2) + $DOUBLE(.1)`.
- If an operation results in a number that is too large to be represented in ObjectScript decimal floating-point (larger than 9.223372036854775807E145), ObjectScript automatically converts this number to $DOUBLE, rather than issuing a <MAXNUMBER> error.

5.3.2 Unary Positive Operator (+)

The unary positive operator (+) gives its single operand a numeric interpretation. If its operand has a string value, it converts it to a numeric value. It does this by sequentially parsing the characters of the string as a number, until it encounters an invalid character. It then returns whatever leading portion of the string was a well-formed numeric. For example:

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

If the string has no leading numeric characters, the unary positive operator gives the operand a value of zero. For example:

```
WRITE + "Thirty-two dollars and 64 cents" // 0
```

The unary positive operator has no effect on numeric values. It does not alter the sign of either positive or negative numbers. For example:

```
SET x = -23
WRITE " x: ", x,! // -23
WRITE "+x: ",+x,! // -23
```

5.3.3 Unary Negative Operator (-)

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

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

If its operand has a string value, the unary negative operator interprets it as a numeric value before reversing its sign. This numeric interpretation is exactly the same as that performed by the unary positive operator, described above. For example:

```
SET x = -23
WRITE -"32 dollars and 64 cents" // -32
```

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.

In the following example, ObjectScript scans the string and encounters the numeric value of 2 and stops there. It then applies the unary negative operator to the value and uses the Concatenate operator (_) to concatenate the value “Rats” from the second string to the numeric value.

```
WRITE -"2Cats"_"Rats" // -2Rats
```

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

5.3.4 Addition Operator (+)

The addition operator produces the sum of two numerically interpreted operands. It uses any leading valid numeric characters as the numeric values of the operands and produces a value that is the sum of the numeric value of the operands.

The following example performs addition on two numeric literals:

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

The following example performs addition on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE "x + y = ",x + y // 9
```

The following example performs string arithmetic on two operands that have leading digits, adding the resulting numerics:

```
WRITE "4 Motorcycles" + "5 bicycles" // 9
```

The following example illustrates that leading zeros on a numerically evaluated operand do not affect the results the operator produces:

```
WRITE "007" + 10 // 17
```

5.3.5 Subtraction Operator (-)

The subtraction operator produces the difference between two numerically interpreted operands. It interprets any leading, valid numeric characters as the numeric values of the operand and produces a value that is the remainder after subtraction.

The following example performs subtraction on two numeric literals:

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

The following example performs subtraction on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE "x - y = ",x - y // -1
```

The following example performs string arithmetic on two operands that have leading digits, subtracting the resulting numerics:

```
WRITE "8 apples" - "4 oranges" // 4
```

If the operand has no leading numeric characters, ObjectScript assumes its value to be zero. For example:

```
WRITE "8 apples" - "four oranges" // 8
```

5.3.6 Multiplication Operator (*)

Binary Multiply produces the product of two numerically interpreted operands. It uses any leading numeric characters as the numeric value of the operands and produces a result that is the product.

The following example performs multiplication on two numeric literals:

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

The following example performs multiplication on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE x * y // 20
```

The following example performs string arithmetic on two operands that have leading digits, multiplying the resulting numerics:

```
WRITE "8 apples" * "4 oranges"  // 32
```

If an operand has no leading numeric characters, Binary Multiply assigns it a value of zero.

```
WRITE "8 apples"*"four oranges" // 0
```

5.3.7 Division Operator (/)

Binary Divide produces the result of dividing two numerically interpreted operands. It uses any leading numeric characters as the numeric value of the operands and products a result that is the quotient.

The following example performs division on two numeric literals:

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

The following example performs division on two locally defined variables:

```
SET x = 4
SET y = 5
WRITE x / y // .8
```

The following example performs string arithmetic on two operands that have leading digits, dividing the resulting numerics:

```
WRITE "8 apples" / "4 oranges"  // 2
```

If the operand has no leading numeric characters, Binary Divide assumes its value to be zero. For example:

```
WRITE "eight apples" / "4 oranges" // 0
// "8 apples"/"four oranges" generates a <DIVIDE> error
```

Note that the second of these operations is invalid. Dividing a number by zero is not allowed. ObjectScript returns a <DIVIDE> error message.

5.3.8 Exponentiation Operator (**)

The Exponentiation Operator produces the exponentiated value of the left operand raised to the power of the right operand.

- `0**0`: Zero raised to the power of zero is 0. However, if either operand is an IEEE double-precision number, (for example, `0**$DOUBLE(0)` or `$DOUBLE(0)**0`) zero raised to the power of zero is 1. For further details, refer to the $DOUBLE function.
- `0**`*n*: 0 raised to the power of any positive number *n* is 0. This includes `0**$DOUBLE("INF")`. Attempting to raise 0 to the power of a negative number results in an error: standard negative numbers generate an <ILLEGAL VALUE> error; **$DOUBLE** negative numbers generate a <DIVIDE> error.
- *num*`**0`: Any non-zero number (positive or negative) raised to the power of zero is 1. This includes `$DOUBLE("INF")**0`.
- `1**`*n*: 1 raised to the power of any number (positive, negative, or zero) is 1.
- `-1**`*n*: -1 raised to the power of zero is 1. -1 raised to the power of 1 or -1 is -1. For exponents larger than 1, see below.

- *num*`**`*n*: A positive number (integer or fractional) raised to any power (integer or fractional, positive or negative) returns a positive number.
- `-`*num*`**`*n*: A negative number (integer or fractional) raised to the power of an even integer (positive or negative) returns a positive number. A negative number (integer or fractional) raised to the power of an odd integer (positive or negative) returns a negative number.
- `-`*num*`**.`*n*: Attempting to raise a negative number to the power of a fractional number results in an <ILLEGAL VALUE> error.
- `$DOUBLE("INF")**`*n*: An infinite number (positive or negative) raised to the power of 0 is 1. An infinite number (positive or negative) raised to the power of any positive number (integer, fractional, or INF) is INF. An infinite number (positive or negative) raised to the power of any negative number (integer, fractional, or INF) is 0.
- `$DOUBLE("NAN")`: NAN on either side of the exponentiation operator always returns NAN, regardless of the value of the other operand.

Very large exponents may result in overflow and underflow values:

- `num**`*nnn*: A positive or negative number greater than 1 with a large positive exponent value (such as `9**153` or `-9.2**152`) generates a <MAXNUMBER> error.
- `num**-`*nnn*: A positive or negative number greater than 1 with a large negative exponent value (such as `9**-135` or `-9.2**-134`) returns 0.
- `.num**`*nnn*: A positive or negative number less than 1 with a large positive exponent value (such as `.22**196` or `-.2**184`) returns 0.
- `.num**-`*nnn*: A positive or negative number less than 1 with a large negative exponent value (such as `.22**-196` or `-.2**-184`) generates a <MAXNUMBER> error.

An exponent that exceeds the maximum value supported by InterSystems IRIS numbers either issues a <MAXNUMBER> error or automatically converts to an IEEE double-precision floating point number. This automatic conversion is specified by using either the **TruncateOverflow()** method of the %SYSTEM.Process class on a per-process basis, or the *TruncateOverflow* property of the Config.Miscellaneous class on a system-wide basis. For further details, refer to the $DOUBLE function.

The following examples performs exponentiation on two numeric literals:

```
WRITE "9 ** 2 = ",9 ** 2,! // 81
WRITE "9 ** -2 = ",9 ** -2,! // .01234567901234567901
WRITE "9 ** 2.5 = ",9 ** 2.5,! // 242.9999999994422343
```

The following example performs exponentiation on two locally defined variables:

```
SET x = 4, y = 3
WRITE "x ** y = ",x ** y,! // 64
```

The following example performs string arithmetic. Exponentiation uses any leading numeric characters as the values of the operands and produces a result.

```
WRITE "4 apples" ** "3 oranges" // 64
```

If an operand has no leading numeric characters, Exponentiation assumes its value to be zero.

The following example demonstrates how to use exponentiation to find the square root of a number.

```
WRITE 256 ** .5 // 16
```

Exponentiation can also be performed using the $ZPOWER function.

5.3.9 Integer Divide Operator (\ )

The Integer Divide operator produces the integer result of the division of the left operand by the right operand. It does not return a remainder, and does not round up the result.

The following example performs integer division on two integer operands. ObjectScript does not return the fractional portion of the number:

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

The following example performs string arithmetic. Integer Divide uses any leading numeric characters as the values of the operands and produces an integer result.

```
WRITE "8 Apples" \ "3.1 oranges" // 2
```

If an operand has no leading numeric characters, ObjectScript assumes its value to be zero. If you attempt integer division with a zero-valued divisor, ObjectScript returns a <DIVIDE> error.

5.3.10 Modulo Operator (#)

The Modulo operator produces the value of an arithmetic modulo operation on two numerically interpreted operands. When the two operands are positive, then the modulo operation is the remainder of the left operand integer divided by the right operand.

The following examples perform modulo operations on numeric literals, returning the remainder:

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

The following example performs string arithmetic. When operating on strings, they are converted to numeric values (according the values described in the section Variable Typing and Conversion) before the modulo operator is applied. Hence, the following two expressions are equivalent:

```
WRITE "8 apples" # "3 oranges",!  // 2
WRITE 8 # 3 // 2
```

Because InterSystems IRIS evaluates a string with no leading numeric characters to zero, a right operand of this kind yields a <DIVIDE> error.

5.4 Logical Comparison Operators

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

5.4.1 Unary Not

Unary Not inverts the truth value of the boolean operand. If the operand is TRUE (1), Unary Not gives it a value of FALSE (0). If the operand is FALSE (0), Unary Not gives it a value of TRUE (1).

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

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

While the following statements produces a result of TRUE (1).

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

Unary Not with a comparison operator inverts the sense of the operation it performs. It effectively inverts the result of the operation. For example, the following statement displays a result of FALSE (0):

```
WRITE 3>5
```

But, the following statement displays a result of TRUE (1):

```
WRITE 3'>5
```

5.4.2 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 Binary Or (!) test in the following program to return TRUE (1):

```
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:

```
SET x=1,y=0
IF (x=1) ! (y=0) {WRITE "TRUE"}
ELSE {WRITE "FALSE" }
// Returns 1 (TRUE)
```

5.4.3 Binary And

Binary 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 Binary And: & 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 evaluates 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).

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

returns 1.

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

returns 1.

The following examples evaluate one true and one false operand and produces a value of FALSE (0).

```
SET A=1,B=0
WRITE "A = ",A,!
WRITE "B = ",B,!
WRITE "A&B = ",A&B,! // FALSE (0)
SET A=1,B=0
WRITE "A&&B = ",A&&B,! // FALSE (0)
```

both return FALSE (0).

The following examples show the difference between the "&" operator and the "&&" operator. In these examples, the left operand evaluates to FALSE (0) and the right operand is not defined. The "&" and "&&" operators respond differently to this situation:

- The "&" operator attempts to evaluate both operands, and fails with an <UNDEFINED> error.

```
TRY {
     KILL B
     SET A=0
     WRITE "variable A defined?: ",$DATA(A),!
     WRITE "variable B defined?: ",$DATA(B),!
     WRITE A&B
     WRITE !,"Success"
     RETURN
    }
CATCH exp
{
    IF 1=exp.%IsA("%Exception.SystemException") {
       WRITE !,"System exception",!
       WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
       WRITE "Data: ",exp.Data,!!
    }
    ELSE { WRITE "not a system exception"}
}
```

- The "&&" operator evaluates only the left operand, and produces a value of FALSE (0).

```
TRY {
     KILL B
     SET A=0
     WRITE "variable A defined?: ",$DATA(A),!
     WRITE "variable B defined?: ",$DATA(B),!
     WRITE A&&B
     WRITE !,"Success"
     RETURN
    }
CATCH exp
{
    IF 1=exp.%IsA("%Exception.SystemException") {
       WRITE !,"System exception",!
       WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
       WRITE "Data: ",exp.Data,!!
    }
    ELSE { WRITE "not a system exception"}
}
```

5.4.3.1 Not And (NAND)

You can specify the Boolean Not And (NAND) operation by using the Unary Not operator with the Binary And (&) operator in either of the following equivalent formats:

```
operand '& operand  '(operand & operand)
```

The Not And operation reverses the truth value of the & Binary And applied to both operands. It produces a value of TRUE (1) when either or both operands are false. It produces a value of FALSE when both operands are TRUE.

The && Binary And operator cannot be prefixed with a Unary Not operator: the format "'&&" is not supported. However, the following format is supported:

```
'(operand && operand)
```

The following example performs two equivalent Not And operations. Each evaluates one FALSE (0) and one TRUE (1) operand and produces a value of TRUE (1).

```
SET A=0,B=1
WRITE !,A'&B   // Returns 1
WRITE !,'(A&B) // Returns 1
```

The following example performs a Not And operation by performing a && Binary And operation and then using a Unary Not to invert the result. The && operation tests the first operand and, because the boolean value is FALSE (0), && does not test the second operand. The Unary Not inverts the resulting boolean value, so that the expression returns TRUE (1):

```
SET A=0
WRITE !,'(A&&B)   // Returns 1
```

5.4.4 Binary Or

Binary 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). Binary Or produces a result of FALSE (0) only if both operands are FALSE (0).

There are two forms to Binary Or: ! (exclamation point) and || (two vertical bars).

- 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 Binary Or to them, and produces a TRUE result:

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

both return TRUE (1).

The following examples evaluate one false and one true operand, apply the Binary Or to them, and produces a TRUE result:

```
SET A=0,B=7
WRITE "A!B = ",A!B,!
SET A=0,B=7
WRITE "A||B = ",A||B,!
```

both return TRUE (1).

The following examples evaluate two false operands and produces a result with a value of FALSE.

```
SET A=0,B=0
WRITE "A!B = ",A!B,!
SET A=0,B=0
WRITE "A||B = ",A||B,!
```

both return FALSE (0).

5.4.4.1 Not Or (NOR)

You can produce a Not Or (NOR) operation by using Unary Not with the ! Binary Or in either of the following equivalent formats:

```
operand '! operand '(operand ! operand)
```

The Not Or operation produces a result of TRUE (1) if both operands have values of FALSE. The Not Or operation produces a result of FALSE (0) if either operand has a value of TRUE or if both operands are TRUE.

The || Binary Or operator cannot be prefixed with a Unary Not operator: the format "'||" is not supported. However, the following format is supported:

```
'(operand || operand)
```

The following Not Or examples evaluate two false operands and produce a TRUE result.

```
SET A=0,B=0
WRITE "A'!B = ",A'!B   // Returns 1

SET A=0,B=0
WRITE "'(A!B) = ",'(A!B)   // Returns 1
```

The following Not Or examples evaluate one TRUE and one false operand and produce a result of FALSE.

```
SET A=0,B=1
WRITE "A'!B = ",A'!B   // Returns 0

SET A=0,B=1
WRITE "'(A!B) = ",'(A!B)   // Returns 0
```

The following Not Or (||) example evaluates the left operand, and because it is TRUE (1) does not evaluate the right operand. The Unary Not inverts the resulting boolean value, so the expression returns FALSE (0).

```
SET A=1
WRITE "'(A||B) = ",'(A||B)   // Returns 0
```

5.5 String Concatenate Operator

The string Concatenate operator (_) is a binary (two-operand) operator that interprets its operands as strings and returns a string value.

You use Concatenate to combine string literals, numbers, expressions, and variables. It takes the form:

operand_operand

Concatenate produces a result that is a string composed of the right operand appended to the left operand. Concatenate gives its operands no special interpretation. It treats them as string values.

The following example concatenates two strings:

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

returns "Highchair".

When concatenating a numeric literal to another numeric literal or to a non-numeric string, InterSystems IRIS first converts each of the numbers to canonical form. The following example concatenates two numeric literals:

```
WRITE 7.00_+008
```

returns 78.

The following example concatenates a numeric literal and a numeric string:

```
WRITE ++7.00_"+007"
```

returns the string 7+007.

The following example concatenates two strings and the null string:

```
SET A="ABC"_""_"DEF"
WRITE A
```

returns “ABCDEF”.

The null string has no effect on the length of a string. You can concatenate an infinite number of null strings to a string.

The maximum string size is 3,641,144 characters. Attempting to concatenate strings that would result in a string exceeding this maximum string size results in a <MAXSTRING> error.

An ObjectScript statement involving multiple concatenations is an atomic (all-or-nothing) operation. In the event of a <MAXSTRING> error, the variable being enlarged by concatenation retains its value prior to the concatenation. For example, if *bigstr* is a string of length 2,000,000, attempting the concatenation `SET bigstr=bigstr_"abc"_bigstr` would result in a <MAXSTRING> error. The length of *bigstr* remains 2,000,000.

5.5.1 Concatenate Encoded Strings

Some ObjectScript strings contain internal encoding that can limit whether these strings can be concatenated:

- A bit string cannot be concatenated, either with another bit string, with a string that is not a bit string, or with the empty string (""). Attempting to do so results in an <INVALID BIT STRING> error when accessing the resulting string.
- A List structure string can be concatenated with another List structure string, or with the empty string (""). It cannot be concatenated with a non-List string. Attempting to do so results in an <LIST> error when accessing the resulting string.
- A JSON string cannot be concatenated, either with another JSON string, with a string that is not a JSON string, or with the empty string (""). Attempting to do so results in an <INVALID OREF> error when accessing the resulting string.

5.6 Numeric Relational Operators

There are two types of relational operators: string relational operators and 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 the section “String-to-Number Conversion”.

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

Comparisons between IEEE double-precision fractional numbers ($DOUBLE numbers) and standard InterSystems IRIS floating point numbers ($DECIMAL numbers) are performed exactly, without rounding. Equality comparisons between $DOUBLE and $DECIMAL numbers often yield unexpected results, and should be avoided. For further details on arithmetic operations involving IEEE double numbers, see the appendix “Numeric Computing in InterSystems Applications” in the *Orientation Guide for Server-Side Programming*.

5.6.1 Binary Less Than

The Binary Less Than operator tests whether its left operand is numerically less than its right operand. ObjectScript evaluates both operands numerically and returns a Boolean result of TRUE (1) is the left operand has a lesser numeric value than its

right operand. ObjectScript returns a Boolean result of FALSE (0) if the left operand has an equal or greater numeric value than the right operand. For example:

```
WRITE 9 < 6
```

returns 0.

```
WRITE 22 < 100
```

returns 1.

5.6.2 Binary Greater Than

Binary Greater Than tests whether the left operand is numerically greater than the right operand. ObjectScript evaluates the two operands numerically and produces a result of TRUE (1) if the left operand is numerically larger than the right operand. It produces a result of FALSE (0) if the left operand is numerically equal to or smaller than the right operand. For example:

```
WRITE 15 > 15
```

returns 0.

```
WRITE 22 > 100
```

returns 0.

5.6.3 Greater Than or Equal To

You can produce a Greater Than or Equal To operation by:

- Combining the Binary Greater Than (>) and Equals (=) operators. The two operators used together give return TRUE if either one returns TRUE.
- Using a Unary NOT operator (') with Binary Less Than (<). The two operators used together reverse the truth value of the Binary Less Than.

ObjectScript produces a result of TRUE (1) when the left operand is numerically greater than or equal to the right operand. It produces a result of FALSE (0) when the left operand is numerically less than the right operand.

You can express the Greater Than or Equal To operation in any of the following ways:

```
operand_A >= operand_B
operand_A '< operand_B
'(operand_A < operand_B)
```

5.6.4 Less Than or Equal To

You can produce a Less Than or Equal To operation by:

- Combining the Binary Less Than (<) and Equals (=) operators. The two operators used together give return TRUE if either one returns TRUE.
- Using a Unary NOT operator (') with Binary Greater Than (>). The two operators used together reverse the truth value of the Binary Greater Than.

ObjectScript produces a result of TRUE (1) when the left operand is numerically less than or equal to the right operand. It produces a result of FALSE (0) when the left operand is numerically greater than the right operand.

You can express the Less Than or Equal To operation in any of the following ways:

```
operand_A <= operand_B
operand_A '> operand_B
'(operand_A > operand_B)
```

The following example tests two variables in a Less Than or Equal To operation. Because both variables have an identical numerical value, the result is TRUE.

```
SET A="55",B="55"
WRITE A'>B
```

returns 1.

5.7 String Relational Operators

There are two types of relational operators: numeric relational operators and 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.

5.7.1 Binary Equals

Binary Equals tests two operands for string equality. When you apply Binary 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:

```
WRITE "SEVEN"="SEVEN"
```

returns TRUE (1).

Binary Equals does not imply any numeric interpretation of either operand. For example, the following statement produces a value of FALSE (0), even though the two operands are numerically identical:

```
WRITE "007"="7"
```

returns FALSE (0).

You can use Binary Equals to test for numeric equality if both operands have a numeric value. For example:

```
WRITE 007=7
```

returns TRUE (1).

You can also force a numeric conversion by using the Unary Arithmetic Positive. For example:

```
WRITE +"007"="7"
```

returns TRUE (1).

If the two operands are of different types, both operands are converted to strings and those strings are compared. Note that this may cause inaccuracy because of rounding and the number of significant digits for conversion to string. For example:

```
WRITE "007"=7,!
 // converts 7 to "7", so FALSE (0)
WRITE 007="7",!
 // converts 007 to "7", so TRUE (1)
WRITE 17.1=$DOUBLE(17.1),!
 // converts both numbers to "17.1", so TRUE (1)
WRITE 1.2345678901234567=$DOUBLE(1.2345678901234567),!
 // compares "1.2345678901234567" to "1.23456789012346", so FALSE (0)
```

5.7.1.1 Not Equals

You can specify a Not Equals operation by using the Unary Not operator with Binary Equals. You can express the Not Equals operation in two ways:

```
operand '= operand
'(operand = operand)
```

Not Equals reverses the truth value of the Binary Equals operator applied to both operands. If the two operands are not identical, the result is TRUE (1). If the two operands are identical, the result is FALSE (0).

5.7.2 Binary Contains

Binary 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.

The following example tests whether L contains S. Because L does contain S, the result is TRUE (1).

```
SET L="Steam Locomotive",S="Steam"
WRITE L[S
```

returns TRUE (1).

The following example tests whether P contains S. Because the character sequence in the strings is different (a period in P and an exclamation point in S), the result is FALSE (0).

```
SET P="Let's play.",S="Let's play!"
WRITE P[S
```

returns FALSE (0).

5.7.2.1 Does Not Contain

You can produce a Does Not Contain operation by using the Unary Not character with Binary Contains in either of the following equivalent formats:

operand A '[*operand B*

'(*operand A* [*operand B*)

The Does Not Contain operation returns TRUE if operand A does not contain the character string represented by operand B and FALSE if operand A does contain the character string represented by operand B. For example,

```
SET P="Beatles", S="Mick Jagger"
WRITE P'[S
```

returns 1.

5.7.3 Binary Follows

Binary Follows tests whether the characters in the left operand come after the characters in the right operand in ASCII collating sequence. Binary Follows tests both strings starting with the left most character in each. The test ends when either:

- A character is found in the left operand that is different from the character at the corresponding position in the right operand.
- There are no more characters left to compare in either of the operands.

ObjectScript returns a value of TRUE if the first unique character in the left operand has a higher ASCII value than the corresponding character in the right operand (that is, if the character in the left operand comes after the character in the right operand in ASCII collating sequence.) If the right operand is shorter than the left operand, but otherwise identical, ObjectScript also returns a value of TRUE.

ObjectScript returns a value of FALSE if any of the following conditions prevail:

- The first unique character in the left operand has a lower ASCII value than the corresponding character in the right operand.
- The left operand is identical to the right operand.
- The left operand is shorter than the right operand, but otherwise identical.

The following example tests whether the string LAMPOON follows the string LAMP in ASCII collation order. The result is TRUE.

```
WRITE "LAMPOON"]"LAMP"
```

returns TRUE (1).

The following example tests whether the string in B follows the string in A. Because BO follows BL in ASCII collation sequence, the result is TRUE.

```
SET A="BLUE",B="BOY"
WRITE B]A
```

returns TRUE (1).

5.7.3.1 Not Follows

You can produce a Not Follows operation by using the Unary Not operator with Binary Follows in either of the following equivalent formats:

```
operand A ']operand B
'(operand A ] operand B)
```

If all characters in the operands are identical or if the first unique character in operand A has a lower ASCII value that the corresponding character in operand B, the Not Follows operation returns a result of TRUE. If the first unique character in operand A has a higher ASCII value that the corresponding character in operand B, the Not Follows operation returns a result of FALSE.

In the following example, because C in CDE does follow A in ABC, the result is FALSE.

```
WRITE "CDE"']"ABC",!
WRITE '("CDE"]"ABC")
```

returns FALSE (0).

5.7.4 Binary Sorts After

Binary 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.

The Binary Sorts After operator returns a TRUE (1) if the first operand sorts after the second and a FALSE (0) if the first operand does not sort after the second. For example:

```
WRITE 122]]2
```

returns TRUE (1).

```
WRITE "LAMPOON"]]"LAMP"
```

returns TRUE (1).

5.7.4.1 Not Sorts After

You can produce a Not Sorts After operation by using the Unary Not operator with Binary Sorts After in either of the following equivalent formats:

```
operand A ']] operand B
'(operand A ]] operand B)
```

If operand A is identical to operand B or if operand B sorts after operand A, then ObjectScript returns a result of TRUE. If operand A sorts after operand B, ObjectScript returns a result of FALSE.

5.8 Pattern Matching

InterSystems IRIS supports two systems of pattern matching:

- ObjectScript pattern matching, described here, a syntax that demarcates the beginning of a pattern string with a question mark (?) or its inverse ('?).
- 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. They are described in a separate chapter of this manual.

These pattern match systems are wholly separate. Each pattern match system can only be used in its own context. It is, however, possible to combine pattern match tests from different pattern match systems using logical AND and OR syntax, as shown in the following example:

```
SET var = "abcDEf"
IF (var ?.e2U.e) && $MATCH(var, "^.{3,7}") { WRITE "It's a match!"}
ELSE { WRITE "No match"}
```

The ObjectScript pattern tests that the string must contain two consecutive uppercase letters. The Regular Expression pattern tests that the string must contain between 3 and 7 characters.

5.8.1 ObjectScript Pattern Matching

The ObjectScript pattern match operator tests whether the characters in its left operand are correctly specified by the pattern in its right operand. It returns a boolean value. The pattern match operator produces a result of TRUE (1) when the pattern correctly specifies the pattern of characters in the left operand. It produces a result of FALSE (0) if the pattern does not correctly specify the pattern of characters in the left 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):

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

The left operand (the test value) and the right operand (the pattern) are distinguished by the leading ? of the right operand. The two operands may be separated by one or more blank spaces, or not separated by blank spaces, as shown in the following equivalent program example:

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

No white space is permitted following the ? operator. White space within the pattern must be within a quoted string and is interpreted as being part of the pattern.

The general format for a pattern match operation is as follows:

```
operand?pattern
```

operand	An expression that evaluates to a string or number, the characters of which you want to test for a pattern.
pattern	A pattern match sequence beginning with a ? character (or with '? for a not-match test). The pattern sequence can be one of the following: a sequence of one or more *pattern-elements*; an indirect reference that evaluates to a sequence of one or more *pattern-elements*

A *pattern-element* consists of one of the following:

- *repeat-count pattern-codes*
- *repeat-count literal-string*
- *repeat-count alternation*

repeat-count	A repeat count — the exact number of instances to be matched. The *repeat-count* can evaluate to an integer or to the period wildcard character (.). Use the period to specify any number of instances.
pattern-codes	One or more pattern codes. If more than one code is specified, the pattern is satisfied by matching any one of the codes.
literal-string	A literal string enclosed in double quotes.
alternation	A set of pattern-element sequences to choose from (in order to perform pattern matching on a segment of the operand string). This provides logical OR capability in pattern specifications.

Use a literal string enclosed in double quotes in a pattern if you want to match a specific character or characters. In other situations, use the special pattern codes provided by ObjectScript. Characters associated with a particular pattern code are (to some extent) locale-dependent. The following table shows the available pattern codes and their meanings:

Table 5–2: Pattern Codes

Code	Meaning
A	Matches any uppercase or lowercase alphabetic character. The 8-bit character set for the locale defines what is an alphabetic character. For the English locale (based on the Latin-1 character set), this includes the ASCII values 65 through 90 (A through Z), 97 through 122 (a through z), 170, 181, 186, 192 through 214, 216 through 246, and 248 through 255.
C	Matches any of the ASCII control characters (ASCII values 0 through 31 and the extended ASCII values 127 through 159).
E	Matches any character, including non-printing characters, whitespace characters, and control characters.
L	Matches any lowercase alphabetic character. The 8-bit character set for the locale defines what is a lowercase character. For the English locale (based on the Latin-1 character set) this includes the ASCII values 97 through 122 (a through z), 170, 181, 186, 223 through 246, and 248 through 255.
N	Matches any of the 10 numeric characters 0 through 9 (ASCII 48 through 57).
P	Matches any punctuation character. The character set for the locale defines what is a punctuation character for an extended (8-bit) ASCII character set. For the English locale (based on the Latin-1 character set), this includes the ASCII values 32 through 47, 58 through 64, 91 through 96, 123 through 126, 160 through 169, 171 through 177, 180, 182 through 184, 187, 191, 215, and 247.
U	Matches any uppercase alphabetic character. The 8-bit character set for the locale defines what is an uppercase character. For the English locale (based on the Latin-1 character set), this includes the ASCII values 65 through 90 (A through Z), 192 through 214, and 216 through 222.
R B M	Matches Cyrillic 8–bit alphabetic character mappings. R matches any Cyrillic character (ASCII values 192 through 255). B matches uppercase Cyrillic characters (ASCII values 192 through 223). M matches lowercase Cyrillic characters (ASCII values 224 through 255). These pattern codes are only meaningful in the Russian 8-bit Windows locale (ruw8). In other locales they execute successfully but fail to match any character.
ZFWCHARZ	Matches any of the characters in the Japanese ZENKAKU character set. ZFWCHARZ matches full-width characters, such as those in the Kanji range, as well as many non-Kanji characters that occupy a double cell when displayed by some terminal emulators. ZFWCHARZ also matches the 303 surrogate pair characters defined in the JIS2004 standard, treating each surrogate pair as a single character. For example, the surrogate pair character $WC(131083) matches ?1ZFWCHARZ. This pattern match code requires a Japanese locale. See the $ZZENKAKU function for further details.
ZHWKATAZ	Matches any of the characters in the Japanese HANKAKU Kana character set. These are Unicode values 65377 (FF61) through 65439 (FF9F). This pattern match code requires a Japanese locale. See the $ZZENKAKU function for further details.

Pattern codes are not case-sensitive; you can specify them in either uppercase or lowercase. For example, ?5N is equivalent to ?5n. You can specify multiple pattern codes to match a specific character or string. For example, ?1NU matches either a number or an uppercase letter.

As stated in the *InterSystems Glossary of Terms*, the ASCII character set refers to an extended, 8-bit character set, rather than the more limited, 7-bit character set.

Note: Pattern matching with double-quote characters can yield inconsistent results, especially when data is supplied from InterSystems IRIS implementations using different NLS locales. The straight double-quote character ($CHAR(34) = ") matches as a punctuation character. Directional double-quote characters (curly quotes) do not match as punctuation characters. The 8-bit directional double-quote characters ($CHAR(147) = “ and $CHAR(148) = ”) match as control characters. The Unicode directional double-quote characters ($CHAR(8220) = “ and $CHAR(8221) = ”) do not match as either punctuation or control characters.

The Pattern Match operator differs from the Binary Contains ([) operator. The Binary Contains operator returns TRUE (1) even if only a substring of the left-hand operand matches the right-hand operand. Also, Binary Contains expressions do not provide the range of options available with the Pattern Match operator. In Binary Contains expressions, you can use only a single string as the right-hand operand, without any special codes.

For example, assume that variable *var2* contains the value “abc”. Consider the following Pattern Match expression:

```
SET match = var2?2L
```

This sets *match* to FALSE (0) because *var2* contains three lowercase characters, not just two.

Here are some examples of basic pattern matching:

```
PatternMatchTest
 SET var = "O"
 WRITE "Is the letter O",!

 WRITE "...an alphabetic character? "
 WRITE var?1A,!

 WRITE "...a numeric character? "
 WRITE var?1N,!

 WRITE "...an alphabetic or ",!,"  a numeric character? "
 WRITE var?1AN,!

 WRITE "...an alphabetic or ",!,"  a ZENKAKU Kanji character? "
 WRITE var?1AZFWCHARZ,!

 WRITE "...a numeric or ",!,"  a HANKAKU Kana character? "
 WRITE var?1ZHWKATAZN
```

You can extend the scope of a pattern code by specifying:

- How many times a pattern can occur
- Multiple patterns
- A combination pattern
- An indefinite pattern
- An alternating pattern

5.8.2 Specifying How Many Times a Pattern Can Occur

To define a range for the number of times that *pattern* can occur in the target operand, use the form:

n.n

The first *n* defines the lower limit for the range of occurrences; the second *n* defines the upper limit.

For the following example, assume that the variable *var3* contains multiple copies of the string “AB” (and no other characters). 1.4 indicates that from one to four occurrences of “AB” are recognized:

```
SET match = var3?1.4"AB"
```

If *var3* = "ABABAB", the expression returns a result of TRUE (1) even though *var3* contains only three occurrences of "AB".

As another example, consider the following expression:

```
SET match = var4?1.6A
```

This expression checks to see whether *var4* contains from one to six alphabetic characters. A result of FALSE (0) is returned if *var4* contains zero or more than six alphabetic characters, or contains a non-alphabetic character.

If you omit either *n*, ObjectScript supplies a default. The default for the first *n* is zero (0). The default for the second *n* is any number. Consider the following example:

```
SET match = var5?.E1"AB".E
```

This example returns a result of TRUE (1) as long as *var5* contains at least one occurrence of the pattern string "AB".

5.8.3 Specifying Multiple Patterns

To define multiple patterns, you can combine *n* and pattern in a sequence of any length. Consider the following example:

```
SET match = date?2N1"/"2N1"/"2N
```

This expression checks for a date value in the format mm/dd/yy. The string "4/27/98" would return FALSE (0) because the month has only one digit. To detect both one and two digit months, you could modify the expression as:

```
SET match = date?1.2N1"/"2N1"/"2N
```

Now the first pattern match (1.2N) accepts either 1 or 2 digits. It uses the optional period (.) to define a range of acceptable occurrences as described in the previous section.

5.8.4 Specifying a Combination Pattern

To define a combination pattern, use the form:

Pattern1Pattern2

With a combination pattern, the sequence consisting of *pattern1* followed by *pattern2* is checked against the target operand. For example, consider the following expression:

```
SET match = value?3N.4L
```

This expression checks for a pattern in which three numeric digits are followed by zero to four lowercase alphabetic characters. The expression returns TRUE (1) only if the target operand contains exactly one occurrence of the combined pattern. For example, the strings "345g" and "345gfij" would qualify, but "345gfijhkbc" "345gfij276hkbc" would not.

5.8.5 Specifying an Indefinite Pattern

To define an indefinite pattern, use the form:

.pattern

With an indefinite pattern, the target operand is checked for an occurrence of *pattern*, but any number of occurrences is accepted (including zero occurrences). For example, consider the expression:

```
SET match = value?.N
```

This expression returns TRUE (1) if the target operand contains zero, one, or more than one numeric character, and contains no characters of any other type.

5.8.6 Specifying an Alternating Pattern (Logical OR)

Alternation allows for testing if an operand matches one or more of a group of specified pattern sequences. It provides logical OR capability to pattern matching.

An alternation has the following syntax:

(*pattern-element sequence* {, *pattern-element sequence* }...)

Thus, the following pattern returns TRUE (1) if *val* contains one occurrence of the letter "A" or one occurrence of the letter "B".

```
SET match = value?1(1"A",1"B")
```

You can have nested alternation patterns, as in the following pattern match expression:

```
SET match = value?.(.(1A,1N),1P)
```

For example, you may want to validate a U.S. telephone number. At a minimum, the phone number must be a 7-digit phone number with a hyphen (-) separating the third and fourth digits. For example:

```
nnn-nnnn
```

The phone number can also include a three-digit area code that must either have surrounding parentheses or be separated from the rest of the number by a hyphen. For example:

```
(nnn) nnn-nnnn
nnn-nnn-nnnn
```

The following pattern match expressions describe three valid forms of a U.S. telephone number:

```
SET match = phone?3N1"-"4N
SET match = phone?3N1"-"3N1"-"4N
SET match = phone?1"("3N1") "3N1"-"4N
```

Without an alternation, the following compound Boolean expression would be required to validate any form of U.S. telephone number.

```
SET match =
   (
   (phone?3N1"-"4N) ||
   (phone?3N1"-"3N1"-"4N) ||
   (phone?1"("3N1") "3N1"-"4N)
   )
```

With an alternation, the following single pattern can validate any form of U.S. telephone number:

```
SET match = phone?.1(1"("3N1") ",3N1"-")3N1"-"4N
```

The alternation in this example allows the area code component of the phone number to be satisfied by either 1"("3N1") " or 3N1"-". The alternation count range of 0 to 1 indicates that the operand *phone* can have 0 or 1 area code components.

Alternations with a repeat count greater than one (1) can produce many combinations of acceptable patterns. The following alternation matches the string shown and matches 26 other three-character strings.

```
SET match = "CAT"?3(1"C",1"A",1"T")
```

5.8.7 Using Incomplete Patterns

If a pattern match successfully describes only part of a string, then the pattern match returns a result of FALSE (0). That is, there cannot be any string left over when the pattern is exhausted. The following expression evaluates to a result of FALSE (0) because the pattern does not match the final "R":

```
SET match = "RAW BAR"?.U1P2U
```

5.8.8 Multiple Pattern Interpretations

There can be more than one interpretation of a pattern as it is matched against an operand. For example, the following expression can be interpreted in two ways:

```
SET match = "/////A#####B$$$$$"?.E1U.E
```

1. The first ".E" matches the substring "/////", the 1U matches the "A", and the second ".E" matches the substring "#####B$$$$$".
2. The first ".E" matches the substring "/////A#####", the 1U matches the character "B", and the second ".E" matches the substring "$$$$$".

   As long as at least one interpretation of the expression is TRUE (1), then the expression has a value of TRUE.

5.8.9 Not Match Operator

You can produce a Not Match operation by using the Unary Not operator (') with Pattern Match:

```
operand'?pattern
```

Not Match reverses the truth value of the Pattern Match. If the characters in the operand cannot be described by the pattern, then Not Match returns a result of TRUE (1). If the pattern matches all of the characters in the operand, then Not Match returns a result of FALSE (0).

The following example uses the Not Match operator:

```
WRITE !,"abc" ?3L
WRITE !,"abc" '?3L
WRITE !,"abc" ?3N
WRITE !,"abc" '?3N
WRITE !,"abc" '?3E
```

5.8.10 Pattern Complexity

A pattern match with multiple alternations and indefinite patterns, when applied to a long string, can recurse many levels into the system stack. In rare cases, this recursion can rise to several thousand levels, threatening stack overflow and a process crash. When this extreme situation occurs, InterSystems IRIS issues a <COMPLEX PATTERN> error rather than risking a crash of the current process.

In the unusual event that such an error occurs, it is recommended that you either simplify your pattern, or apply it to shorter subunits of the original string.

You can interrupt pattern execution by issuing a **Crtl-C** key command, resulting in an <INTERRUPT> error.

5.9 Indirection

The ObjectScript indirection operator (@) allows you to assign values indirectly to variables. 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. InterSystems IRIS performs the substitution before execution of the associated command.

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.

You should use indirection only in those cases where it offers a clear advantage. Indirection can have an impact on performance because InterSystems IRIS performs the required evaluation at runtime, rather than during the compile phase. Also, if you use complicated indirections, be sure to document your code clearly. Indirections can sometimes be difficult to decipher.

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.

```
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
```

InterSystems IRIS recognizes five types of indirection:

- *Name indirection*
- *Pattern indirection*
- *Argument indirection*
- *Subscript indirection*
- *$TEXT argument indirection*

Which type of indirection is performed depends on the context in which the *@variable* occurs. Each type of indirection is described separately below.

Indirection cannot be used with dot syntax. This is because dot syntax is parsed at compile time, not at runtime.

5.9.1 Name Indirection

In name indirection, the indirection evaluates to a variable name, a line label, or a routine name. InterSystems IRIS substitutes the contents of *variable* for the expected name before executing the command.

Name indirection can only access public variables. For further details, refer to the User-defined Code chapter of this manual.

When you use indirection to reference a named variable, the value of the indirection must be a complete global or local variable name, including any necessary subscripts. In the following example, InterSystems IRIS sets the variable B to the value of 6.

```
SET Y = "B",@Y = 6
```

Important: If a call attempts to use indirection to get or set the value of object properties, it may result in an error. Do not use calls of this kind, as they attempt to bypass property accessor methods (**<PropertyName>Get** and **<PropertyName>Set**). Instead, use the **$CLASSMETHOD**, **$METHOD**, and **$PROPERTY**) functions, which are designed for this purpose; see the section "Dynamically Accessing Objects" in the "Using Objects with ObjectScript" chapter of *Defining and Using Classes* for more information.

When you use indirection to reference a line label, the value of the indirection must be a syntactically valid line label. In the following example, InterSystems IRIS sets D to:

- The value of the line label FIG if the value of N is 1.
- The value of the line label GO if the value of N is 2.
- The value of STOP in all other cases.

Later, InterSystems IRIS passes control to the label whose value was given to D.

```
B SET D = $SELECT(N = 1:"FIG",N = 2:"GO",1:"STOP")
 ; ...
LV GOTO @D
```

When you use indirection to reference a routine name, the value of the indirection must be a syntactically valid routine name. In the following example, name indirection is used on the **DO** command to supply the appropriate procedure name. At execution time, the contents of variable *loc* are substituted for the expected name:

```
Start
 READ !,"Enter choice (1, 2, or 3): ",num
 SET loc = "Choice"_num
 DO @loc
 RETURN
Choice1()
 ; ...
Choice2()
 ; ...
Choice3()
 ; ...
```

Name indirection can substitute only a name value. The second **SET** command in the following example returns an error message because of the context. When evaluating the expression to the right of the equal sign, InterSystems IRIS interprets *@var1* as an indirect reference to a variable name, not a numeric value.

```
 SET var1 = "5"
 SET x = @var1*6
```

You can recast the example to execute correctly as follows:

```
 SET var1 = "var2",var2 = 5
 SET x = @var1*6
```

5.9.2 Pattern Indirection

Pattern indirection is a special form of indirection. The indirection operator replaces a pattern match. The value of the indirection must be a valid pattern. (Pattern matching is described under "Pattern Matching".) Pattern indirection is especially useful when you want to select several possible patterns and then use them as a single pattern.

In the following example, indirection is used with pattern matching to check for a valid U.S. Postal (ZIP) code. Such codes can take either a five-digit (*nnnnn*) or a nine-digit (*nnnnnnnnn*) form.

The first **SET** command sets the pattern for the five-digit form. The second **SET** command sets the pattern for the nine-digit form. The second **SET** command is executed only if the postconditional expression `($LENGTH(zip) = 10)` evaluates to TRUE (nonzero), which occurs only if the user inputs the nine digit form.

```
GetZip()
 SET pat = "5N"
 READ !,"Enter your ZIP code (5 or 9 digits): ",zip
 SET:($LENGTH(zip)=10) pat = "5N1""-""4N"
 IF zip'?@pat {
   WRITE !,"Invalid ZIP code"
   DO GetZip()
 }
```

The use of indirection with pattern matching is a convenient way to localize the patterns used in an application. In this case, you could store the patterns in separate variables and then reference them with indirection during the actual pattern tests. (This is also an example of name indirection.) To port such an application, you would have to modify only the pattern variables themselves.

5.9.3 Argument Indirection

In argument indirection, the indirection evaluates to one or more command arguments. By contrast, name indirection applies only to part of an argument.

To illustrate this difference, compare the following example with the example given under Name Indirection.

```
Start
 SET rout = "^Test1"
 READ !,"Enter choice (1, 2, or 3): ",num
 SET loc = "Choice"_num_rout
 DO @loc
 QUIT
```

In this case, @loc is an example of argument indirection because it supplies the complete form of the argument (that is, *label^routine*). In the name indirection example, @loc is an example of name indirection because it supplies only part of the argument (the *label* name, whose entry point is assumed to be in the current, rather than a separate, routine).

In the following example, the second **SET** command is an example of name indirection (only part of the argument, the name of the variable), while the third **SET** command is an example of argument indirection (the entire argument).

```
 SET a = "var1",b = "var2 = 3*4"
 SET @a = 5*6
 SET @b
 WRITE "a = ",a,!
 WRITE "b = ",b,!
```

5.9.4 Subscript Indirection

Subscript indirection is an extended form of name indirection. In subscript indirection, the value of the indirection must be the name of a local or global array node. Subscript indirection is syntactically different than the other forms of indirection. Subscript indirection uses two indirection operators in the following format:

```
@array@(subscript)
```

Assume that you have a global array called *^client* in which the first-level node contains the client's name, the second-level node contains the client's street address, and the third-level node contains the client's city, state, and ZIP code. To write out the three nodes for the first record in the array, you can use the following form of the **WRITE** command:

```
 WRITE !,^client(1),!,^client(1,1),!,^client(1,1,1)
```

When executed, this command might produce output similar to following:

```
John Jones
42 Arnold St.
Boston, MA 02745
```

To write out a range of records (say, the first 10), you could modify the code so that the **WRITE** is executed within a **FOR** loop. For example:

```
 FOR i = 1:1:10 {
 WRITE !,^client(i),!,^client(i,1),!,^client(i,1,1)
 }
```

As the **FOR** loop executes, the variable i is incremented by 1 and used to select the next record to be output.

While more generalized than the previous example, this is still very specialized code because it explicitly specifies both the array name and the number of records to output.

To transform this code into a more generalized form that would allow a user to list a range of records from any array (global or local) that stores name, street, and city information in three node levels, you could use subscript indirection as shown in the following example:

```
Start
 READ !,"Output Name, Street, and City info.",!
 READ !,"Name of array to access: ",name
 READ !,"Global or local (G or L): ",gl
 READ !,"Start with record number: ",start
 READ !,"End with record number: ",end
 IF (gl["L")!(gl["l") {SET array = name}
 ELSEIF (gl["G")!(gl["g") {SET array = "^"_name}
 SET x = 1,y = 1
 FOR i = start:1:end {DO Output}
 RETURN
Output()
 WRITE !,@array@(i)
 WRITE !,@array@(i,x)
 WRITE !,@array@(i,x,y)
 QUIT
```

The **WRITE** commands in the Output subroutine use subscript indirection to reference the requested array and the requested range of records.

In the evaluation of subscript indirection, if the instance of indirection refers to an unsubscripted global or local variable, the value of the indirection is the variable name and all characters to the right of the second Indirection operator, including the parentheses.

For a local variable, the maximum number of subscript levels is 255. Subscript indirection cannot reference more than 254 subscripts for a multidimensional object property. For a global variable, the maximum number of subscript levels depends on the subscript, and may be higher than 255, as described in Global Structure in *Using Globals*. Attempting to use indirection to populate a local variable with more than 255 subscript levels results in a <SYNTAX> error.

A class parameter can be used as the base for subscript indirection in the same way that a local or global variable can be used as the base. For example, you can perform subscript indirection using a class parameter with the following syntax:

```
SET @..#myparam@(x,y) = "stringval"
```

5.9.5 $TEXT Argument Indirection

As its name implies, **$TEXT** argument indirection is allowed only in the context of a $TEXT function argument. The value of the indirection must be a valid **$TEXT** argument.

You use **$TEXT** argument indirection primarily as a convenience to avoid multiple forms of indirection that produce the same result. For example, if the local variable LINE contains the entry reference " START^MENU", you can use name indirection to the line label and to the routine name to obtain the text for the line, as follows:

```
SET LINETEXT = $TEXT(@$PIECE(LINE,"^",1)^@$PIECE(LINE,"^",2))
```

You can use **$TEXT** argument indirection to produce the same result in a simpler manner, as follows:

```
SET LINETEXT = $TEXT(@LINE)
```

6

Regular Expressions

InterSystems IRIS® data platform supports regular expressions for use with the following ObjectScript functions $LOCATE and $MATCH and methods of the %Regex.Matcher class.

All other substring matching operations use the InterSystems IRIS Pattern Matching operators.

This chapter describes the following features of regular expressions:

- Wildcard and Quantifiers. Example: `.*` matches any number of characters of any type.
- Literals and Character Ranges. Example: `[A-Z]` matches a single uppercase character in the range A through Z.
- Character Type Meta-Characters are sequences that match a group of characters:
  - Single-letter Character Types. Example: `\d` matches any digit character.
  - Unicode Property Character Types. Example: `\p{LL}` matches any lowercase letter.
  - POSIX Character Types. Example: `[:print:]` matches any printable character.
- Grouping Construct uses parentheses to repeatedly apply a regular expression. Example: `(\p{LL})+` checks each character to determine if it is a lowercase letter.
- Anchors that limit where a match can occur. Example: `\b(day)` matches only those occurrences of “day” that occur at a word boundary.
- Logical Operators. Example: `[[:upper:]&&[:greek:]]` matches uppercase Greek letters.
- Character Representation Meta-Characters are sequences that match a single character.
  - Hexadecimal, Octal, and Unicode Representation. Example: `\x5A` is the hexadecimal representation for the letter Z.
  - Control Character Representation. Example: `\cM` is the carriage return control character.
  - Symbol Name Representation. Example: `\N{equals sign}` is the = character.
- Modes. Example: `(?i)` makes all subsequent matches not case-sensitive.
- Comments. Example: `(?# date and 24-hour time)` inserts this comment into the regular expression string.
- Error Messages.

InterSystems IRIS implementation of regular expressions is based on the International Components for Unicode (ICU) standard for regular expressions. Users familiar with Perl regular expressions will find many similarities to the InterSystems IRIS implementation.

6.1 Wildcard and Quantifiers

.	Wildcard. Matches any single character of any type, except the line spacing characters $CHAR(10), $CHAR(11), $CHAR(12), $CHAR(13), and $CHAR(133). This exclusion of line spacing characters can be overridden by specifying (?s) single-line mode (as described later in this reference page). Can be used alone ".." = any two characters, or in combination "\d.." = a digit character followed by any two characters of any type. Can be combined with suffixes (with the same line spacing characters restriction): `.?` = zero or one character of any type. `.*` = zero or more characters of any type. `.+` = one or more characters of any type. `.{3}` = exactly 3 characters of any type. To end a wildcard sequence, you escape the next literal by using the backslash (\) prefix. For example, the *regexp* `".*\H\d{2}"` matches a string of any characters of any type that ends with the letter "H" followed by a two-digit number.
?	Single-character suffix (0 or 1). Applies *regexp* 1 or 0 times to *string*. The regular expressions "\d?", "[0–9]?", or "[[:digit:]]?" all match to either a single number or the empty string. The regular expression ".?(log)" can match "blog" (1 occurrence) or "log" (0 occurrences). The regular expression "abc?" can match either "abc" or "ab".
+	Repetition suffix (1 or more). Applies *regexp* one or more times to *string*. For example, "A+" matches the string "AAAAA". ".+" matches a string of any length of any character type, but does not match the empty string. The regular expressions "\d+", "[0–9]+", or "[[:digit:]]+" all match a string of numbers of any length. You can use parentheses for complex repeating patterns. For example, (AB)+" matches the string "ABABABAB"; "(\d\d\d\s)+" matches a sequence of any length of three numbers alternating with a single blank space.
*	Repetition suffix (0 or more). Applies *regexp* zero, one, or more than one times to *string*. For example, "A*" matches the strings "A", "AAAAA", and the empty string. ".*" matches a string of any length of any character type, including the empty string. The regular expressions "\d*", "[0–9]*", or "[[:digit:]]*" all match a string of numbers of any length or the empty string. You can use parentheses for complex repeating patterns. For example, (AB)*" matches the string "ABABABAB"; "(\d\d\d\s)*" matches a sequence of any length of three numbers alternating with a single blank space.
{*n*}	Quantification suffix (*n* times). The `{n}` suffix applies *regexp* exactly *n* number of times. For example, "\d{5}" matches any number with five digits.
{*n*,}	Quantification suffix (at least *n* times). The `{n,}` suffix applies *regexp n* or more times. For example, "\d{5,}" matches any number with five or more digits.
{*n*,*m*}	Quantification suffix (range). The `{n,m}` suffix applies *regexp* a minimum of *n* times and a maximum of *m* times (inclusive). For example, "\d{7,10}" matches any number of at least 7 digits but not more than 10 digits.

6.2 Literals and Character Ranges

Most literal characters can simply be included in a regular expression. For example, the regular expression `".*G.*"` specifies that the string must contain the letter G.

Some literal characters are also used as regular expression meta-characters. You must use the escape prefix (the backslash character) before a meta-character that is to be treated as a literal character. The following literal characters require an escape prefix: dollar sign `\$`; asterisk `\*`; plus sign `\+`; period `\.`; question mark `\?`; backslash `\\`; caret `\^`; vertical bar `\|`; open and close parentheses `\( \)`; open and close square brackets `\[ \]`; open and close curly braces `\{ \}`. The close square bracket `]` does not always require an escape prefix; the escape prefix should be used for clarity and consistency.

The quote character does not take an escape prefix; to specify a literal quote character, double it `""`.

The following are ways to specify more than one regular expression match for a literal:

[*x*]	A specified character or list of characters. Thus [A] means that only the uppercase letter character “A” is a match, and [ACE] matches any one of the letters A, C, or E. Characters may be listed in any sequence. Repeated characters are permitted. You can use a caret (^) to specify the inverse; for example, [^A] means that any character *except* “A” is a match; [^XYZ] means that any character except X, Y, or Z is a match. By default, these character matches are case-sensitive. You can make character matching not case-sensitive by preceding it with the (?i) mode modifier. To specify a caret (^) as a literal match character it cannot be the first character in the list. To specify a hyphen ($CHAR(45)) as a literal match character it must be the first or last character in the list. To specify a close bracket (]) as a literal match character it must be the first character in the list. (First character can mean the first character after the ^ inverse operator). Backslash escape prefix literals can also be used; for example [\\AB\[CD] matches backslash (\), open bracket ([), and the letters A, B, C, and D.
[*x*-*z*]	A range of specified characters beginning with *x* and ending with *z* (inclusive). Though commonly used for letters or numbers, any ascending ASCII sequence can be used as a range. Thus [A-Z] is the range for all uppercase letters. [A-z] is a range that includes not only all uppercase and lowercase letters, but the six ASCII punctuation characters between the alphabets. Specifying a range that is not in ascending ASCII sequence generates a <REGULAR EXPRESSION> error. You can also specify multiple ranges. Thus [A-Za-z] is the range for all uppercase and lowercase letters. You can use a caret (^) as the first character after the open bracket to specify the inverse; for example, [^A-F] means all character *except* A through F. The caret specifies the inversion of all of the specified ranges; thus [^A-Za-z] means any character except a letter. Ranges of characters and lists of single characters can be combined in any sequence. Thus [ABCa-fXYZ0-9] matches the characters specified and the characters within the specified ranges.
(*str*) (*str1*|*str2*)	A specified string or a list of strings separated by the OR logical operator (|). Thus (William) matches this exact substring in *string*, and (William|Willy|Wm\.|Bill) matches any of these substrings. You can use the escape prefix | to specify a vertical bar as a literal within a string. By default, these substring matches are case-sensitive. You can make a substring match not case-sensitive by preceding it with the (?i) mode modifier. By default, these substring matches can occur anywhere in *string*. You can restrict substring matching to occurrences at a word boundary by preceding it with \b.

6.3 Character Type Meta-Characters

InterSystems IRIS regular expressions support three sets of character type meta-characters:

- Single-letter character types. For example: \d
- Unicode property character types. For example: \p{LL}
- POSIX character types. For example [:alpha:]

These character type meta-characters can be used in any regular expression in any combination.

6.3.1 Single-letter Character Types

A single-letter character type meta-character is indicated by the backslash (\) character, followed by a letter. The character type is specified by a lowercase letter (\d = a digit: 0 through 9). For those character types that support inversion, an uppercase letter specifies the inverse of the character type (\D = any character except a digit).

\a	A bell character $CHAR(7). No inverse is supported.
\d	A digit character. The numbers 0 through 9. The inverse is \D.
\e	An escape character $CHAR(27). No inverse is supported.
\f	A form feed character $CHAR(12). No inverse is supported.
\n	A newline character $CHAR(10). No inverse is supported.
\r	A carriage return character $CHAR(13). No inverse is supported.
\s	A spacing character. A blank space, a tab, or a line spacing character, including the following characters: $CHAR(9), $CHAR(10), $CHAR(11), $CHAR(12), $CHAR(13), $CHAR(32), $CHAR(133), and $CHAR(160). The inverse is \S.
\t	A tab character $CHAR(9). No inverse is supported.
\w	A word character. A word character can be a letter, a number, or the underscore character. Valid letters include uppercase and lowercase letters, including Unicode letters. They include the following extended ASCII characters: $CHAR(170), $CHAR(181), $CHAR(186), $CHAR(192) through $CHAR(214), $CHAR(216) through $CHAR(246), $CHAR(248) through $CHAR(256). The inverse is \W.

The \d, \s, and \w meta-characters also match appropriate Unicode characters beyond $CHAR(256).

For meta-character sequences for other individual control characters, see Control Character Representation.

6.3.2 Unicode Property Character Types

Unicode property character type matching matches a single character to a character type specified using the following syntax:

```
\p{prop}
```

For example, \p{LL} matches any lowercase letter. A *prop* keyword consists of one or two letter characters; *prop* keywords are not case-sensitive. The single-letter *prop* keywords are the most inclusive; two-letter *prop* keywords specify a subset.

The inverse is \P{*prop*}. For example, \P{LL} matches any character that is *not* a lowercase letter.

The following table shows the characters that match each *prop* keyword for the first 256 characters (an example Unicode character is provided for the *prop* keywords that do not match any of the 256 characters):

C: control and miscellaneous characters 0–31, 127–159, 173	CC: control characters 0–31, 127–159	CF: formatting characters 173	CN: unassigned code points (for example, 888)	CO: private use characters (for example, 57344)	CS: surrogates (for example, 55296)
L: letters 65-90, 97–122, 170, 181, 186, 192–214, 216–246, 248–255	LL: lowercase letters 97–122, 170, 181, 186, 223–246, 248–255	LM: modifier letters (for example, 688)	LO: other letters not LL, LU, LT, or LM (for example, 443)	LT: titlecase letters (for example 453)	LU: uppercase letters 65-90, 192–214, 216–222
M: marks (for example, 768)	MC: modification characters (for example, 2307)	ME: marks that enclose (for example, 1160)	MN: accent marks (for example, 768)		
N: numbers 48–57, 178–179, 185, 188–190	ND: decimal numbers 48–57	NL: letters representing numbers (for example, 5870)	NO: number subscripts and fractions 178–179, 185, 188–190		
P: punctuation 33–35, 37–42, 44–47, 58–59, 63–64, 91–93, 95, 123, 125, 161, 171, 183, 187, 191	PC: connecting punctuation 95	PD: dashes 45	PE: closing punctuation 41, 93, 125 PS: opening punctuation 40, 91, 123	PI: initial punctuation 171 PF: final punctuation 187	PO: other punctuation 33–35, 37–39, 42, 44, 46–47, 58–59, 63–64, 92, 161, 183, 191
S: symbols 36, 43, 60–62, 94, 96, 124, 126, 162–169, 172, 174–177, 180, 182, 184, 215, 247	SC: currency symbols 36, 162–165	SK: combining symbols 94, 96, 168, 175, 180, 184	SM: math symbols 43, 60–62, 124, 126, 172, 177, 215, 247	SO: other symbols 166–167, 169, 174, 176, 182	
Z: separators 32, 160	ZL: line separators (for example, 8232)	ZP: paragraph separators (for example, 8233)	ZS: space characters 32, 160		

You can use the following code to determine which characters match with a *prop* keyword:

```
READ prop#2:10
READ rangefrom:10
READ rangeto:10
FOR i=rangefrom:1:rangeto {
    IF $MATCH($CHAR(i),"\p{"_prop_"}")=1 {
       WRITE i,"=",$CHAR(i),!} }
```

6.3.3 POSIX Character Types

POSIX syntax matches a single character to a character type specified by a *ptype* keyword using either of the following syntax forms:

```
\p{ptype}
[:ptype:]
```

For example, `[:lower:]` or `\p{lower}` matches any lowercase letter. You can specify the inverse (match anything except a lowercase letter) as follows: `[:^lower:]` or `\P{lower}`.

The *ptype* keywords are not case-sensitive. The general *ptype* keywords are:

- alnum — letters and numbers.
- alpha — letters.
- blank — the tab $CHAR(9) or space $CHAR(32), $CHAR(160).
- cntrl — control characters: $CHAR(0) through $CHAR(31), $CHAR(127) through $CHAR(159).
- digit — the numbers 0 through 9.
- graph — printable characters, excluding the space character: $CHAR(33) thorough $CHAR(126), $CHAR(161) thorough $CHAR(156).
- lower — lowercase letters.
- math — mathematics characters (a subset of symbol). Includes the following characters: `+<=>^|~¬±×÷`
- print — printable characters, including the space character: $CHAR(32) thorough $CHAR(126), $CHAR(160) thorough $CHAR(156).
- punct — punctuation characters (excludes symbol characters). Includes the following characters: `!"#%&'()*,-./:;?@[\]_{}¡«·»¿`
- space — spacing characters, including the blank space, tab, and line spacing characters, including the following characters: $CHAR(9), $CHAR(10), $CHAR(11), $CHAR(12), $CHAR(13), $CHAR(32), $CHAR(133), and $CHAR(160).
- symbol — symbol characters (excludes punctuation characters). Includes the following characters: `$+<=>^`|~¢£¤¥¦§¨©¬®¯°±´¶¸×÷`
- upper — uppercase letters.
- xdigit — hexadecimal digits: the numbers 0 through 9, the uppercase letters A through F, the lowercase letters a through f.

In addition, you can use *ptype* to specify a Unicode category. For example, `[:greek:]` matches any character in the Unicode Greek category (this includes the Greek letters which are found in the range $CHAR(900) through $CHAR(974)). A partial list of these POSIX Unicode categories includes: `[:arabic:]`, `[:cyrillic:]`, `[:greek:]`, `[:hebrew:]`, `[:hiragana:]`, `[:katakana:]`, `[:latin:]`, `[:thai:]`. These Unicode categories can also be represented as `[:script=greek:]`, for example.

The following example uses POSIX matching to compare the `[:letter:]` character set and the `[:latin:]` character set in the first 256 characters. They differ by a single character, $CHAR(181):

```
FOR i=0:1:255 {
  SET letr="foo"
  IF 1=$MATCH($CHAR(i),"[:letter:]") {
   SET letr=$CHAR(i)}
  IF 1=$MATCH($CHAR(i),"[:latin:]") {
       SET lat=$CHAR(i)}
   ELSE {SET lat="foo"}
  IF letr '= lat {WRITE i," ",$CHAR(i),!}
}
```

6.4 Grouping Construct

You can use parentheses to specify a literal or meta-character sequence applied repeatedly. For example, the regular expression `([0–9])+` tests each successive character in a string to determine if it is a number.

This usage is shown in the following examples:

```
WRITE $MATCH("4567683285759","([0-9])+"),!
    // test for all numbers, no empty string
WRITE $MATCH("4567683285759","([0-9])*"),!
    // test for all numbers or for empty string
WRITE $MATCH("Now is the time","\p{LU}(\p{L}|\s)+"),!
    // test for initial uppercase letter, then all letters or spaces
WRITE $MATCH("MAboston-9a","\p{LU}{2}(\p{LL}|\d|\-)*"),!
    // test for 2 uppercase letters, then all lowercase, numbers, dashes, or ""
WRITE $MATCH("1^23^456^789","([0-9]+\^?)+"),!
    // test for one or more numbers followed by 0 or 1 ^ characters, apply test repeatedly
WRITE $MATCH("$1,234,567,890.99","\$([0-9]+,?)+\.\d\d")
   // test for $, then numbers followed by 0 or 1 comma, then decimal point, then 2 fractional digits
```

Note: Because grouping constructs apply a regular expression repeatedly, it is possible to create a matching operation that takes a long time to complete.

The following cautionary example shows how the execution time for a repeatedly applied grouping construct increases rapidly depending on the position of the pattern match error in the string. The more permutations that must be tested before declaring a non-match, the longer the execution time:

```
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,2222222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("11111x11111,2222222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,22x22222222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,2222222x222,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b,!
SET a=$ZHOROLOG
  WRITE $MATCH("1111111111,22222222x22,3333333333","([0-9]+,?)+")
  SET b=$ZHOROLOG-a
  WRITE " duration: ",b
```

6.5 Anchor Meta-Characters

An anchor is a meta-character that limits the regular expression match associated with it to a particular place in the match string. For example, a match can only occur at the beginning or end of the string, or after a space character in the string.

6.5.1 String Beginning or End

These anchors limit matching to the beginning or end of the string.

^ \A	Beginning of string anchor prefix. Indicates that the regular expression match must occur at the beginning of the string.
$	End of string anchor suffix. Indicates that the regular expression match must occur at the end of the string. End-of-line characters (ASCII 10, 11, 12, or 13) are ignored. Same as \Z.
\Z	End of string anchor suffix. Indicates that the regular expression match must occur at the end of the string. End-of-line characters (ASCII 10, 11, 12, or 13) are ignored. Same as $.
\z	End of string anchor suffix. Indicates that the regular expression match must occur at the end of the string. End-of-line characters (ASCII 10, 11, 12, or 13) are treated as string characters for matching.

The following example shows how a beginning of string anchor limits a **$LOCATE** match:

```
SET str="ABCDEFG"
WRITE $LOCATE(str,"A"),!   // returns 1
WRITE $LOCATE(str,"D"),!   // returns 4
WRITE $LOCATE(str,"^A"),!  // returns 1
WRITE $LOCATE(str,"^D"),!  // returns 0 (no match)
```

The following example shows how an end of string anchor limits a **$LOCATE** match:

```
SET str="ABCDABCD"
WRITE $LOCATE(str,"(ABC)"),!   // returns 1
WRITE $LOCATE(str,"D"),!       // returns 4
WRITE $LOCATE(str,"(ABC)$"),!  // returns 0 (no match)
WRITE $LOCATE(str,"(ABCD)$"),! // returns 5
WRITE $LOCATE(str,"D$"),!      // returns 8
```

The following example shows how end-of-string anchors handle a line feed character:

```
SET str="ABCDEFG"_$CHAR(10)

WRITE $LOCATE(str,"G$"),!                  // returns 7
WRITE $LOCATE(str,"G"_$CHAR(10)_"$"),!     // returns 7
WRITE $LOCATE(str,$CHAR(10)_"$"),!!        // returns 8

WRITE $LOCATE(str,"G\Z"),!                 // returns 7
WRITE $LOCATE(str,"G"_$CHAR(10)_"\Z"),!    // returns 7
WRITE $LOCATE(str,$CHAR(10)_"\z"),!!       // returns 8

WRITE $LOCATE(str,"G\z"),!                 // returns 0
WRITE $LOCATE(str,"G"_$CHAR(10)_"\z"),!    // returns 7
WRITE $LOCATE(str,$CHAR(10)_"\z"),!        // returns 8
```

6.5.2 Word Boundary

You can limit matching to occurrences at a word boundary. A word boundary is identified by a word character next to a non-word character, or a word character at the beginning of the string. Word characters are those that match the \w character type: letters, numbers, and the underscore character. Commonly, this is the first letter(s) of a word at the beginning of *string* or following a space character or other punctuation. The regular expression syntax for a word boundary is:

- \b matches an occurrence at a non-word character/word character boundary, or a word character at the beginning of a string.
- \B (the inverse) matches an occurrence at a word character/word character boundary, or at a non-word character/non-word character boundary.

The following example use \b to match word boundaries that begin with the substring “in” or “un”:

```
SET str(1)="unlucky"          // match: "un" is at start of string
SET str(2)="highly unlikely"  // match: "un" follows a space character
SET str(3)="fall in place"    // match: "in" can be followed by a space
SET str(4)="the %integer"     // match: % is a non-word character
SET str(5)="down-under"       // match: - is a non-word character
SET str(6)="winning"          // no match: "in" preceded by word character
SET str(7)="the 4instances"   // no match: a number is a word character
SET str(8)="down_under"       // no match: an underscore is a word character
FOR i=1:1:8 {
    WRITE $MATCH(str(i),".*\b[iu]n.*")," string",i,!
    }
```

The following example uses \B to locate the regular expression when it is *not* at a word boundary:

```
SET str(1)="the thirteenth item"
WRITE $LOCATE(str(1),"\Bth")   // returns 13 ("th" preceded by a word character)
SET str(2)="the^thirteenth^item"
```

The following example show how \b and \B can be used in a regular expression that does not specify a word character:

```
SET str(1)="this##item"
WRITE $LOCATE(str(1),"\b#"),!   // returns 5 (the first # at a word boundary)
WRITE $LOCATE(str(1),"\B#")     // returns 6 (the first # not at a word boundary)
```

6.6 Logical Operators

You can represent compound character types by combining values with logical AND (&&), logical OR (|), and subtract (– –) operators. A compound character type must be enclosed in square brackets.

Implicit OR: You can use square brackets without logical operators to specify lists or ranges of matching characters, one of which must be true. The following examples match all uppercase letters and the numbers 1234: `[\p{LU}1234]` or `[[:upper:]1234]`, `[\p{LU}1-4]` or `[[:upper:]1-4]`.

AND (&&): You can use logical AND to specify multiple character type meta-characters, both of which must be true. For example, to limit a match to only uppercase Greek letters, you could specify: `[\p{LU}&&\p{greek}]` or `[[:upper:]&&[:greek:]]`.

OR (|): You can use logical OR to specify multiple character type meta-characters, either of which must be true. For example, to limit a match to either numbers or Greek letters, you could specify: `[\p{N}|\p{greek}]` or `[[:digit:]|[:greek:]]`. Note that this use of an explicit OR is optional; a list of character types without logical operators is interpreted as logical OR.

SUBTRACT (– –): You can use logical subtract to specify multiple character type meta-characters, the first of which must be true and the second of which must be false. For example, to limit a match all uppercase letters *except* Greek letters, you could specify: `[\p{LU}--\p{greek}]` or `[[:upper:]--[:greek:]]`.

6.7 Character Representation Meta-Characters

The following are meta-character representations of individual characters. Each sequence matches with a single character.

Note that a few individual control characters ($CHAR(7), $CHAR(9), $CHAR(10), $CHAR(12), $CHAR(13), and $CHAR(27)) can also be represented using a single-letter character type.

6.7.1 Hexadecimal, Octal, and Unicode Representation

\x*nn* \x{*nnn*}	Hexadecimal representation. For example, \x5A is the letter 'Z'. Note that the hex letters A through F are not case-sensitive. Leading zeros can be included or omitted. \x*nn* can be used for one-digit or two-digit hexadecimal numbers. For hexadecimal numbers with more digits you must use the \x{*nnn*} curly brace syntax, where *nnn* can be from 1 to 7 hex digits, with a maximum value of 010FFFF. For example, \x{005A} is the letter 'Z', \x{396} is the Greek letter zeta.
\0*nnn*	Octal representation. The *nnn* value is an octal value of two, three, or four digits; however, the leftmost digit must be a zero. For example, the carriage return character $CHAR(13) can be represented by `\015` or `\0015`. The maximum value is `\0377`, which is $CHAR(255).
\u*nnnn*	Unicode representation. The *nnnn* value is a four-digit hexadecimal number corresponding to the Unicode character. For example, `\u005A` is the letter 'Z' ($CHAR(90); `\u03BB` is the Greek lowercase lambda ($CHAR(955)).

6.7.2 Control Character Representation

Control characters are the non-printing ASCII characters $CHAR(0) through $CHAR(31). They can be represented using the following syntax:

```
\cX
```

where *X* is a letter or symbol that corresponds to an ASCII control character (characters 0 through 31). Letters correspond to $CHAR(1) through $CHAR(26). For example, \cH is $CHAR(8), the backspace character. An *X* letter is not case-sensitive. The non-letter control characters follow the same ASCII character set sequence, as follows: $CHAR(0) = \c@ or \c`, $CHAR(27) = \c{ or \c[, $CHAR(28) = \c| or \c\, $CHAR(29) = \c} or \c], $CHAR(30) = \c^ or \c~, $CHAR(31) = \c_.

6.7.3 Symbol Name Representation

This character type can be used to match single printable punctuation, space, and symbol characters. The syntax is as follows:

```
\N{charname}
```

For example, `\N{comma}` matches a comma. Note that the meta-character \N must be an uppercase letter.

The supported character names include: acute accent (´), ampersand (&), apostrophe ('), asterisk (*), breve (˘), cedilla (¸), colon (:), comma (,), dagger (†), degree sign (°), division sign (÷), dollar sign ($), double dagger (‡), em dash (—), en dash (–), exclamation mark (!), equals sign (=), full stop (.), grave accent (`), infinity ( ), left curly bracket ({), left parenthesis ((), left square bracket ([), macron (¯), multiplication sign (×), plus sign (+), pound sign (#), prime ( ), question mark (?), right curly bracket (}), right parenthesis ()), right square bracket (]), semicolon (;), space ( ), square root ( ), tilde (~), vertical line (|). Also supported are subscript zero though subscript nine and superscript zero though superscript nine.

6.8 Modes

A mode changes the interpretation of the character matches that follows it. The *mode* is specified by a single lowercase letter. There are two ways to use modes:

- Mode for a regular expression sequence. For example: (?i)
- Mode for a specified literal within a regular expression. . For example: (?i:(fred|ginger))

The following *mode* characters are supported:

(?i)	Case mode. When active, letter case is disregarded when matching uppercase and lowercase letters to a regular expression.
(?m)	Multi-line mode. Affects the behavior of ^ (beginning of string) and $ (end of string) anchors, when applied to a multi-line string. By default these anchors apply to the entire string. When multi-line mode is active, these anchors apply to the beginning and end of each line within a multi-line string. A line can be begun by any of the newline characters: 10, 11, 12, 13, 133 (and Unicode 8232 and 8233).
(?s)	Single-line mode. When off, the dot (.) wildcard does not match the newline characters: 10, 11, 12, 13, 133 (and Unicode 8232 and 8233). When on, the dot (.) wildcard matches all characters, including newline characters. Note that the pair of characters carriage return ($CHAR(13)) and line feed ($CHAR(10)), when specified in that order, are counted in a regular expression as a single character.
(?x)	Free-spacing mode. Allows for whitespace and trailing comments in a regular expression.

6.8.1 Mode for a Regular Expression Sequence

A regexp mode governs regular expression interpretation from the point where it is applied to the end of the regular expression, or until explicitly turned off. The syntax is as follows:

```
(?n)  to turn mode on
(?-n) to turn mode off
```

Where *n* is a single lowercase letter that specifies the mode type.

The following example shows case mode (?i):

```
WRITE $MATCH("A","(?i)[abc]"),!
WRITE $MATCH("a","(?i)[abc]")
```

The following example shows case mode (?i). The first regular expression is case-sensitive. The second regular expression begins with the case mode modifier (?i) makes the regular expression not case-sensitive:

```
SET name(1)="Smith,John"
SET name(2)="dePaul,Lucius"
SET name(3)="smith,john"
SET name(4)="John Smith"
SET name(5)="Smith,J"
SET name(6)="R2D2,CP30"
SET n=1
WHILE $DATA(name(n)) {
  IF $MATCH(name(n),"\p{LU}\p{LL}+,\p{LU}\p{LL}+")
    { WRITE name(n)," : case match",! }
  ELSEIF $MATCH(name(n),"(?i)\p{LU}\p{LL}+,\p{LU}\p{LL}+")
    { WRITE name(n)," : non-case match",! }
  ELSE { WRITE name(n)," : not a valid name",! }
  SET n=n+1 }
```

The following example shows single-line mode (?s), which allows `".*"` to match a string containing newline characters:

```
SET line(1)="This is a string without line breaks."
SET line(2)="This is a string with"_$CHAR(10)_"one line break."
SET line(3)="This is a string"_$CHAR(11)_"with"_$CHAR(12)_"two line breaks."
SET i=1
WHILE $DATA(line(i)) {
  IF $MATCH(line(i),".*") {WRITE "line(",i,") is a single line string",! }
  ELSEIF $MATCH(line(i),"(?s).*") {WRITE "line(",i,") is a multiline string",! }
  ELSE {WRITE "string error",! }
  SET i=i+1 }
```

The following example shows in single-line mode (?s) that the carriage return/line feed pair (in that order) are counted in a regular expression as one character:

```
SET str(1)="one"_$CHAR(13)_$CHAR(10)_"two"   // CR/LF
SET str(2)="one"_$CHAR(10)_$CHAR(13)_"two"   // LF/CR
SET i=1
WHILE $DATA(str(i)) {
   WRITE $LENGTH(str(i))," is the length of string ",i,!
   IF $MATCH(str(i),"(?s).{7}") { WRITE "string ",i," matches 7 chars",! }
   ELSEIF $MATCH(str(i),"(?s).{8}") { WRITE "string ",i," matches 8 chars",! }
   ELSE { WRITE "string match error",! }
   SET i=i+1
 }
```

The following example shows multi-line mode (?m). It locates the substring identified by the end anchor ($). In single-line mode, this end substring is always “break”, the last substring in the string. In multi-line mode the end substring can be any of the substrings that end a line within a multi-line string:

```
SET line(1)="String without line break"
SET line(2)="String with"_$CHAR(10)_" one line break"
SET line(3)="String"_$CHAR(11)_" with"_$CHAR(12)_" two line break"
SET i=1
WHILE $DATA(line(i)) {
  WRITE $LOCATE(line(i),"(String|with|break)$")," line(",i,") in single-line mode",!
  WRITE $LOCATE(line(i),"(?m)(String|with|break)$")," line(",i,") in multi-line mode",!!
  SET i=i+1 }
```

6.8.2 Mode for a Literal

You can also apply a mode modifier to a literal (or a set of literals), using the syntax:

```
(?mode:literal)
```

This mode modification applies just to the literal(s) within the parentheses.

The following case mode (?i) example matches last names (*lname*) that begin with the de, del, dela, and della, regardless of the capitalization of this prefix. The rest of *lname* must begin with a capital letter, followed by at least one lowercase letter:

```
SET lname(1)="deTour"
SET lname(2)="DeMarco"
SET lname(3)="DeLaRenta"
SET lname(4)="DelCarmine"
SET lname(5)="dellaRobbia"
SET i=1
WHILE $DATA(lname(i)) {
   WRITE $MATCH(lname(i),"(?i:de|del|dela|della)\p{LU}\p{LL}+")," = ",lname(i),!
   SET i=i+1 }
```

6.9 Comments

Within a regular expression you can specify two types of comments:

- Embedded comments
- Line end comment (in (?x) mode only)

6.9.1 Embedded Comments

You can include embedded comments within a regular expression by using the following syntax:

```
(?# comment)
```

The following example show the use of comments within a regular expression to document that this format match is for an American format date (MM/DD/YYYY), not a European format date (DD/MM/YYYY):

```
WRITE $MATCH("04/28/2012","^[01]\d(?# months)/[0123]\d(?# days)/\d\d\d\d$")
```

6.9.2 Line End Comment

When free-spacing mode (?x) is in effect, you can include a comment at the end of a regular expression using the following syntax:

```
# comment
```

The following example shows an end comment in free-spacing mode:

```
WRITE $MATCH("04/28/2012","^[01]\d/[0123]\d/\d\d\d\d$")," no comment",!
WRITE $MATCH("04/28/2012","^[01]\d/[0123]\d/\d\d\d\d$# date test")," comment no (?x) mode",!
WRITE $MATCH("04/28/2012","(?x)^[01]\d/[0123]\d/\d\d\d\d$# date test")," comment in (?x) mode",!
```

In free-spacing mode, whitespace can be included within the regular expression.

6.10 Error Messages

An improperly specified *regexp* generates a <REGULAR EXPRESSION> error. To determine the type of error, you can invoke the **LastStatus()** method, as shown in the following example:

```
TRY {
  WRITE "TRY block:",!
  WRITE $MATCH("A","\p{LU}"),!  // good regexp
  WRITE $MATCH("A","\p{}"),!    // bad regexp
}
CATCH exp {
  WRITE !,"CATCH block exception handler:",!
  IF 1=exp.%IsA("%Exception.SystemException") {
    WRITE "System exception",!
    WRITE "Name: ",$ZCVT(exp.Name,"O","HTML"),!
    WRITE "Location: ",exp.Location,!
    WRITE "Code: ",exp.Code,!! }
  ELSE {WRITE "Unexpected exception type",!  RETURN }
  WRITE "%Regex.Matcher status:"
  DO $SYSTEM.Status.DisplayError(##class(%Regex.Matcher).LastStatus())
  RETURN
}
```

For a list of these errors, refer to General Error Messages 8300 through 8352 in the *InterSystems IRIS Error Reference*.

7

Commands

The *command* is the basic unit of code in ObjectScript programming on InterSystems IRIS® data platform. 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 chapter describes the following aspects of ObjectScript commands:

- Command Keywords
- Command Arguments
- Postconditionals
- Multiple Commands on a Line

The chapter then briefly describes the following groups of ObjectScript commands:

- Commands governing variable assignment — SET, KILL, and NEW
- Context commands for the execution of other commands — **TRY** and **CATCH** error handling structure; **TSTART**, **TCOMMIT**, and **TROLLBACK** transaction processing; **LOCK** resource locking
- Commands that invoke code (and exit code) — DO, JOB, and XECUTE; QUIT and RETURN
- Flow of control commands — IF, ELSEIF, and ELSE; FOR; WHILE and DO WHILE
- Input/Output commands — the four display (write) commands; READ; OPEN, USE, and CLOSE

This is not a complete list of ObjectScript commands. These and many other ObjectScript commands are described in detail in the *ObjectScript Reference*.

7.1 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:

```
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 in the *ObjectScript Reference*.

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.

7.2 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:

```
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:

```
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:

```
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, refer to White Space in the “Syntax Rules” chapter of this manual.

7.2.1 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:

```
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 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:

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

but the following command is not valid:

```
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:

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

7.2.2 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.

```
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.

7.2.3 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:

```
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.

7.2.3.1 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**.

  ```
  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**:

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

7.3 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 postcon-

ditionals 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.

7.3.1 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.

7.3.2 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*:

```
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",!
}
```

7.4 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 refer to the General System Limits appendix to the *Orientation Guide for Server-Side Programming*. Note that if you are using Studio to write or edit source code, this limit may differ.

7.5 Variable Assignment Commands

To provide the necessary functionality for variable management, ObjectScript provides the following commands:

- SET assigns a value to a variable.
- KILL deletes the assignment of a value to a variable.
- NEW establishes a new context for variable assignment.

7.5.1 SET

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

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

```
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:

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

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

```
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:

```
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:

```
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:

```
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:

```
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, refer to the SET command in the *ObjectScript Reference*.

7.5.2 KILL

The **KILL** command deletes variables from memory and can be used to delete them from disk. Its basic form is:

```
KILL expression
```

where *expression* is one or more variables to delete. The simplest forms of **KILL**, then, are:

```
KILL x
KILL x,y,z
```

A special form of **KILL**, called an "exclusive **KILL**", deletes all local variables *except* those specified. To use an exclusive **KILL**, place its argument in parentheses. For example, if you have variables *x*, *y*, and *z*, you can delete *y*, *z*, and any other local variables *except x* by invoking:

```
KILL (x)
```

Without any arguments, **KILL** deletes all local variables.

For further details, refer to the KILL command in the *ObjectScript Reference*.

7.5.3 NEW

The **NEW** command creates a new local variable context. This means that it preserves the existing local variable value in the old context, then initiates a new context in which the local variable is not assigned a value. In an application that uses procedures, use **NEW** to initialize variables for the entire application or a major subsystem of the application.

The following syntax forms are supported:

```
NEW          // initiate new context for all local variables
NEW x        // initiate new context for the specified local variable
NEW x,y,z    // initiate new context for the listed local variables
NEW (y)      // initiate new context for all local variables except the specified variable
NEW (y,z)    // initiate new context for all local variables except the listed variables
```

For further details, refer to the NEW command in the *ObjectScript Reference*.

7.6 Code Execution Context Commands

The following commands are used to support the execution of groups of commands:

- **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. The **TRY** block contains multiple commands performing a desired operation. Each **TRY** block is paired with a **CATCH** error handling block that is invoked when an error occurs in the **TRY** block. Each **TRY** block must be immediately followed by its **CATCH** block. You can create multiple **TRY** / **CATCH** block pairs in a program, if desired. You can also nest **TRY** / **CATCH** block pairs. You can use the **THROW** command from within a **TRY** block to explicitly invoke the corresponding **CATCH** block. For further details, refer to The TRY-CATCH Mechanism in the "Error Processing" chapter of this manual. Refer to the TRY, THROW, and CATCH commands in the *ObjectScript Reference*.
- **TSTART**, **TCOMMIT**, and **TROLLBACK** commands for transaction processing. It is recommended that when a group of commands should be executed atomically (as a single all-or-nothing unit of code) you begin the group of commands with a **TSTART** command and end it with a **TCOMMIT** command. If a problem occurs during execution, you should issue a **TROLLBACK** command to roll back the operations performed by the group of commands. See the Transaction Processing chapter for more information. Refer to the TSTART, TCOMMIT, and TROLLBACK commands in the *ObjectScript Reference*.
- The **LOCK** command for locking and unlocking resources. See the Transaction Processing chapter for more information. Refer to the LOCK command in the *ObjectScript Reference*.

7.7 Invoking Code

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

- DO
- JOB
- XECUTE
- QUIT and RETURN

7.7.1 DO

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

```
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,

```
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:

```
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:

```
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, refer to the DO command in the *ObjectScript Reference*.

7.7.2 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, refer to the JOB command in the *ObjectScript Reference*.

7.7.3 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, refer to the XECUTE command in the *ObjectScript Reference*.

7.7.4 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, refer to the QUIT and RETURN commands in the *ObjectScript Reference*.

7.8 Flow Control Commands

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

7.8.1 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, refer to the Operators and Expressions chapter of this manual.

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:

7.8.1.1 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:

```
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 IF command in the *ObjectScript Reference*.

7.8.2 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:

```
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:

```
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:

```
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.

```
FOR b = "John", "Paul", "George", "Ringo" {
   WRITE !, "Was ", b, " the leader? "
 READ choice
}
```

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:

```
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**:

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

For further details, refer to the FOR command in the *ObjectScript Reference*.

7.8.3 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**:

```
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, refer to the WHILE command and the DO WHILE command in the *ObjectScript Reference*.

7.9 I/O Commands

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

7.9.1 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

7.9.1.1 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.

7.9.1.2 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 7–1: 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 7–2: 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, refer to the WRITE, ZWRITE, ZZDUMP, and ZZWRITE commands in the *ObjectScript Reference*.

7.9.2 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:

```
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, refer to the READ command in the *ObjectScript Reference*.

7.9.3 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 I/O Device Guide.

For further details, refer to the OPEN, USE, and CLOSE commands in the *ObjectScript Reference*.

8

Callable User-defined Code Modules

This chapter describes how to create and invoke user-defined modules of ObjectScript code on InterSystems IRIS® data platform. These units of code can be user-defined functions, procedures, methods, routines, or subroutines.

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:

```
ProcedureName(Parameters) [PublicVariables]
 {
    /* code goes here */
    QUIT 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 the section "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 the section "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 **QUIT** 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.

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

This chapter 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.

Important: By default, methods are procedures. Because of this, all content on procedures in this chapter also describes methods. For more information about methods, see the chapter "Methods" in the book Defining and Using Classes.

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.

8.1.1 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:

```
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:

```
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.

8.1.2 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:

```
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.

8.1.3 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 **QUIT** statement. It can accept parameters and can also return a value. To invoke a function, there are two valid forms of the syntax:

```
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:

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

If you enclose the code and **QUIT** 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:

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

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

```
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
   QUIT c
```

The code invoking the function can ignore the return value, but a function's **QUIT** command must specify a return value. Attempting to exit a function with an argumentless **QUIT** 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 **QUIT** 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.

8.2 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":

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

8.2.1 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.

8.2.1.1 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.

8.2.1.2 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.

8.2.2 Procedure Syntax

Procedure syntax:

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

Invoking syntax:

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

or

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

where

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.
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 with a **QUIT**. 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 “*/”.

8.2.3 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):

```
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
    }

USER>WRITE

USER>DO ^publicvarsexample

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

a=1
b=2
USER>
```

8.2.3.1 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 with a **QUIT**.

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 "%" 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.

8.2.3.2 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 we **QUIT** the procedure, and also it protects any previous value that public variable may have had. For example, the code:

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

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.

8.2.3.3 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,

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

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

8.2.4 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,

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

defines a public procedure, while

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

and

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

both define a private procedure.

8.3 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:

```
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:

```
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.

8.3.1 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** 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.

```
 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,!
 QUIT 1
 }
Cube(num) PUBLIC {
 SET result = num*num*num
 QUIT result
}
```

8.3.2 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. Passing by reference is the only way you can pass an array name to a procedure. Note that 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.

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:

```
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,!
  QUIT
  }
```

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:

```
Start ; Raise an integer to a power.
 READ !,"Integer= ",num  QUIT:num=""
 READ !,"Power= ",powr   QUIT: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}
 QUIT z
```

8.3.3 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:

```
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)`:

```
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,!
  QUIT
  }
```

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:

```
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,!
  QUIT
  }
```

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:

```
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,!
  QUIT 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:

```
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:

```
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
  }
```

For specifics about methods that can accept a variable number of parameters, see the section "Variable Numbers of Arguments in Methods" in the "Methods" chapter of *Defining and Using Classes*.

8.4 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:

  ```
  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:

```
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:
  - **$TEXT** supports label+offset from the procedure label.
  - **GOTO** label+offset is supported in direct mode lines from the procedure label as a means of returning to the procedure following a **Break** or error.
  - The **ZBREAK** command supports a specification of label+offset from the procedure label.
  - The **$TEST** state in effect when the procedure was called is restored upon the **QUIT** for the procedure.
  - 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.

8.5 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:

```
 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:

```
    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.

8.6 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:

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

For more information on error traps, see the chapter of this document on Error Processing.

8.7 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.

8.7.1 Subroutines

8.7.1.1 Syntax

Routine syntax:

```
label [ ( param [ = default  ][ , ...] ) ]
 /* code */
 QUIT
```

Invoking syntax:

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

or

```
GOTO label
```

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 "falling through."

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.
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.

8.7.1.2 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.

8.7.2 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.

8.7.2.1 Syntax

Non-procedure function syntax:

```
label([param[=default]][,...])
   code
   QUIT expression
```

Invoking syntax:

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

or

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

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.

8.7.2.2 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:

```
SET x=$$myfunc()
```

8.7.2.3 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.

8.7.2.4 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.

8.7.2.5 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:

  ```
  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.

8.7.2.6 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.

8.7.2.7 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:

  ```
  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.

9

ObjectScript Macros and the Macro Preprocessor

The ObjectScript compiler includes a preprocessor and ObjectScript includes support for preprocessor directives. These directives allow you to create macros for use in applications — both in methods and routines. These macros provide the functionality for simple textual substitutions in code. InterSystems IRIS® data platform itself also includes various predefined macros, which are described in the relevant contexts within the documentation set.

Preprocessor directives are included in the MAC version of code. When you compile MAC code, the ObjectScript compiler performs preprocessing and generates INT (intermediate, readable ObjectScript) code and OBJ (executable object) code.

This chapter has three parts:

- Using Macros — Describes how to define and use macros.
- Preprocessor Directives Reference — Provides a list of the preprocessor directives available for use in macro definitions.
- Using System-supplied Macros — Describes using predefined macros that are available with InterSystems IRIS and provides a list of them.

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.

The following globals return MAC code information. 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 file. 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.

9.1 Using Macros

This section covers the following topics:

- Creating Custom Macros
- Saving Custom Macros
- Invoking Macros
- Referring to External Macros (Include Files)

9.1.1 Creating Custom Macros

Macros are one-line definitions of substitutions that can support many aspects of ObjectScript functionality. In their basic form, they are created with a #Define directive. For instance, the following code creates a macro called `StringMacro` and makes it a substitution for the string "Hello, World!":

```
#Define StringMacro "Hello, World!"
```

(You can use ##Continue to continue a #Define directive to the next line.)

ObjectScript allows you to invoke a macro using the "$$$" syntax, such as:

```
  WRITE $$$StringMacro
```

which, in this case, displays the string "Hello, World!" Here is the entire sample:

```
#Define StringMacro "Hello, World!"
   WRITE $$$StringMacro
```

Supported functionality includes:

- String substitutions, as demonstrated above.
- Numeric substitutions:

  ```
  #Define NumberMacro 22

  #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:

  ```
  #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:

  ```
  #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:

  ```
  #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:

```
#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.

9.1.1.1 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 the Pattern Matching section of the "Operators and Expressions" chapter of this book.
- Macro names should not begin with ISC, because ISC*name*.inc files are reserved for system use.

9.1.1.2 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.

9.1.1.3 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 the "Comments" section in the "Syntax Rules" chapter of *Using ObjectScript* for basic information on 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:

```
/// 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 "Referring to External Macros (Include Files)." For information on Studio Assist, see the "Editor Options" section of the "Setting Studio Options" chapter of *Using Studio*.

9.1.2 Saving Custom Macros

Macros can either appear in the file where they are invoked or, as is more common, in a separate include file. For a macro to be available class-wide (that is, available for any member to invoke), place its definition in an include file and include it in the class.

To place macros in an include file, the procedure in Studio is:

1. Select **Save** or **Save As** from the **File** menu.
2. In the **Save As** dialog, specify that the **Files of type** field has a value of **Include File (*.inc)**.

   Note: The **Macro Routine (*.mac)** value for this field is not the correct file type for ObjectScript macros.

3. Enter the directory and file names, and save the macros.

9.1.3 Invoking Macros

You can invoke a macro either when its definition is part of the method or routine being defined or when that method or routine uses the **#Include** directive to refer to a definition in an external source. For information on **#Include**, see the section Referring to External Macros or the reference section on #Include.

To invoke a macro from within ObjectScript code, refer to it by its name prefixed with "$$$". Hence, if you have defined a macro called `MyMacro`, you can call it by referring to `$$$MyMacro`. Note that macro names are case-sensitive.

You can invoke a macro to substitute a value in contexts where you cannot supply a variable value. For example:

```
#Define StringMacro "Hello",!,"World!"
   WRITE $$$StringMacro

#Define myclass "Sample.Person"
  SET x=##class($$$myclass).%New()
```

Remember that macros are text substitutions. After a macro is substituted in, the syntax for the resulting statement is checked for correctness. 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.

9.1.4 Referring to External Macros (Include Files)

When you have saved macros to a separate file, you can make them available with the **#include** directive. The directive is not case-sensitive, so it can appear as **#Include**.

When including macros within a class or at the beginning of a routine, the directives are of the form:

```
#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**.

For example, if you have one or more macros in the file MyMacros.inc, you can include them with the following call:

```
#include MyMacros
```

If there are other macros that are in the file YourMacros.inc, you can include all of them with the following calls:

```
#include MyMacros
#include YourMacros
```

When you use #include in a routine you must specify a separate #include statement on a separate line for each include file.

To include an include file at the beginning of a class definition, the syntax is of the form:

```
include MyMacros
```

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`. Also, compilation in Studio converts the form of the word so that its first letter is capitalized and subsequent letters are lower case.

The ObjectScript compiler provides a `/defines` qualifier that permits including external macros. For further details refer to the Compiler Qualifiers table in the **$SYSTEM** special variable reference page in the *ObjectScript Reference*.

See also the reference section on #Include.

9.2 Preprocessor Directives Reference

InterSystems IRIS includes support for the following system preprocessor directives:

- #;
- #Def1Arg
- #Define
- #Dim
- #Else
- #ElseIf
- #EndIf
- #Execute
- #If
- #IfDef
- #IfNDef
- #Import
- #Include
- #NoShow
- #Show
- #SQLCompile Audit
- #SQLCompile Mode

- #SQLCompile Path
- #SQLCompile Select
- #UnDef
- ##;
- ##Continue
- ##Expression
- ##Function
- ##Lit
- ##Quote
- ##QuoteExp
- ##SQL
- ##StripQ
- ##Unique

Note: The macro preprocessor directives are not case-sensitive. This document displays their names in title case for clarity, but this is not required.

9.2.1 #;

The **#;** preprocessor directive creates a single line comment that does not appear in .int code. The comment appears only in either .mac code or in an include file. The **#;** appears at the beginning (column 1) of the line. The comment continues for the remainder of the current line. It has the form:

```
#; Comment here...
```

where the comment follows the "#;".

#; makes an entire line a comment. Compare with the ##; preprocessor directive, which makes the rest of the current line a comment.

9.2.2 #Def1Arg

The **#Def1Arg** preprocessor directive defines a macro with only one argument, where that argument can have commas in it. **#Def1Arg** is a special version of **#Define**, since **#Define** treats commas *within* arguments as delimiters *between* arguments. It has the form:

```
#Def1Arg Macro(%Arg) Value
```

where

- *Macro* is the name of the macro being defined, which accepts only one argument. A valid macro name is an alphanumeric string.
- *%Arg* is the name of the argument for the macro. The name of the variable specifying the macro's argument must begin with a percent sign.
- *Value* is the macro's value, which includes the use of value of %Arg specified at runtime

A **#Def1Arg** line can include a ##; comment.

For more information on defining macros generally, see the entry for **#Define**.

For example, the following `MarxBros` macro accepts the comma-separated list of the names of the Marx brothers as its argument:

```
#Def1Arg MarxBros(%MBNames) WRITE "%MBNames:",!,"The Marx Brothers!",!
 // some movies have all four of them
 $$$MarxBros(Groucho, Chico, Harpo, and Zeppo)
 WRITE !
 // some movies only have three of them
 $$$MarxBros(Groucho, Chico, and Harpo)
```

where the `MarxBros` macro takes an argument, *%MBNames* argument, which accepts a comma-delimited list of the names of the Marx brothers.

9.2.3 #Define

The **#Define** preprocessor directive defines a macro. It has the form:

```
#Define Macro[(Args)] [Value]
```

where

- *Macro* is the name of the macro being defined. A valid macro name is an alphanumeric string.
- *Args* (optional) are the one or more arguments that it accepts. These are of the form `(arg1, arg2, ...)`. The name of each variable specifying a macro argument must begin with a percent sign. Argument values cannot include commas.
- *Value* (optional) is the value being assigned to the macro, where the value can be any valid ObjectScript code. This can be something as simple as a literal or as complex as an expression.

If a macro is defined with a value, then that value replaces the macro in ObjectScript code. If a macro is defined without a value, then code can use other preprocessor directives to test for the existence of the macro and then perform actions accordingly.

You can use ##Continue to continue a **#Define** directive to the next line. You can use ##; to append a comment to a **#Define** line. But you cannot use ##Continue and ##; on the same line.

9.2.3.1 Macros with Values

Macros with values provide a mechanism for simple textual substitutions in ObjectScript code. Wherever the ObjectScript compiler encounters the invocation of a macro (in the form `$$$MacroName`), it replaces the value specified for the macro at the current position in ObjectScript code. The value of a macro can be any valid ObjectScript code. This includes:

- A string
- A numeric value
- A class property
- The invoking of a method, function, or other code

Macro arguments cannot include commas. If commas are required, the **#Def1Arg** directive is available.

The following are examples of definitions for macros being used in various ways.

```
#Define Macro1 22
#Define Macro2 "Wilma"
#Define Macro3 x+y
#Define Macro4 $Length(x)
#Define Macro5 film.Title
#Define Macro6 +$h
#Define Macro7 SET x = 4
#Define Macro8 DO ##class(%Library.PopulateUtils).Name()
#Define Macro9 READ !,"Name: ",name  WRITE !,"Nice to meet you, ",name,!

#Define Macro1A(%x) 22+%x
#Define Macro2A(%x) "Wilma" _ ": %x"
#Define Macro3A(%x) (x+y)*%x
#Define Macro4A(%x) $Length(x) + $Length(%x)
#Define Macro5A(%x) film.Title _ ": " _ film.%x
#Define Macro6A(%x) +$h - %x
#Define Macro7A(%x) SET x = 4+%x
#Define Macro8A(%x) DO ##class(%Library.PopulateUtils).Name(%x)
#Define Macro9A(%x) READ !,"Name: ",name  WRITE !,"%x ",name,!
#Define Macro9B(%x,%y) READ !,"Name: ",name  WRITE !,"%x %y",name,!
```

Conventions for Macro Values

Though a macro can have *any* value, the convention is a macro is literal expression or complete executable line. For example, the following is valid ObjectScript syntax:

```
#Define Macro7 SET x =
```

where the macro might be invoked with code such as:

```
 $$$Macro7 22
```

which the preprocessor would expand to

```
 SET x = 22
```

Though this is clearly valid ObjectScript syntax, this use of macros is discouraged.

9.2.3.2 Macros without Values

A macro can be defined without a value. In this case, the existence (or not) of the macro specifies that a particular condition exists. You can then use other preprocessor directives to test if the macro exists and perform actions accordingly. For example, if an application is compiled either as a Unicode executable or an 8-bit executable, the code might be:

```
#Define Unicode

#IfDef Unicode
   // perform actions here to compile a Unicode
   // version of a program
#Else
   // perform actions here to compile an 8-bit
   // version of a program
#EndIf
```

9.2.3.3 JSON Escaped Backslash Restriction

A macro should not attempt to accept a JSON string containing the `\"` escape convention. A macro value or argument cannot use the JSON `\"` escape sequence for a literal backslash. This escape sequence is not permitted in the body of a macro or in the formal arguments passed to a macro expansion. As an alternative, the `\"` escape can be changed to `\u0022`. This alternative works for JSON syntax strings used as both key names and element values. In the case where the JSON string containing a literal backslash is used as an element value of a JSON array or a JSON object, you can alternatively replace the JSON string containing `\"` with an ObjectScript string expression enclosed in parentheses that evaluates to the same string value.

9.2.4 #Dim

The **#Dim** preprocessor directive specifies the intended type of a local variable. It has the form:

```
#Dim VariableName As DataTypeName
#Dim VariableName As DataType = InitialValue
#Dim VariableName As List Of DataType
#Dim VariableName As Array Of DataType
```

where:

- *VariableName* is the variable being defined, or a comma-separated list of variables.
- *DataType* is the type of *VariableName*. Specifying a data type is optional, you can omit As DataType and just specify =InitialValue.
- *InitialValue* is a value optionally specified for *VariableName*. (This syntax is not available for lists or arrays.)

If *VariableName* specifies a comma-separated list of data variables, all the variables are assigned the same data type and initial value. For example:

```
#Dim a,b,c As %String = "default string"
```

If *VariableName* specifies a comma-separated list of object variables, each variable is assigned a separate OREF. For example:

```
#Dim d,e,f As %DynamicArray = ["element1","element2"]
#Dim g,h,i As Sample.Person = ##class(Sample.Person).%New()
```

For further details, see Variable Declaration and Scope in the "Variables" chapter of this manual.

9.2.5 #Else

The **#Else** preprocessor directive specifies the beginning of the fall-through case in a set of preprocessor conditions. It can follow **#IfDef**, **#If**, or **#ElseIf**. It is followed by **#EndIf**. It has the form:

```
#Else
  // subsequent indented lines for specified actions
#EndIf
```

The #Else directive keyword should appear on a line by itself. Anything following #Else on the same line is considered a comment and is not parsed.

For an example of **#Else** with **#If**, see that directive; for an example with **#EndIf**, see that directive.

Note: If **#Else** appears in method code and has an argument other than a literal value of 0 or 1, the compiler generates code in subclasses (rather than invoking the method in the superclass). To avoid generating this code, test conditions for a value of 0 or 1, which results in simpler code and optimizes performance.

9.2.6 #ElseIf

The **#ElseIf** preprocessor directive specifies the beginning of a secondary case in a set of preprocessor conditions that begin with **#If**. Hence, it can follow **#If** or another **#ElseIf**. It is followed by another **#ElseIf**, **#Else** or **#EndIf**. (The **#ElseIf** directive is not for use with **#IfDef** or **#IfNDef**.) It has the form:

```
#ElseIf <expression>
  // subsequent indented lines for specified actions

  // next preprocessor directive
```

where <*expression*> is a valid ObjectScript expression. If <*expression*> evaluates to a non-zero value, it is true.

Any number of spaces may separate #ElseIf and <*expression*>. However, no spaces are permitted within <*expression*>. Anything following <*expression*> on the same line is considered a comment and is not parsed.

For an example, see **#If**.

Note: **#ElseIf** has an alternate name of **#ElIf**. The two names behave identically.

9.2.7 #EndIf

The **#EndIf** preprocessor directive concludes a set of preprocessor conditions. It can follow **#IfDef**, **#IfUnDef**, **#If**, **#ElseIf**, and **#Else**. It has the form:

```
  // #IfDef, #If, or #Else specifying the beginning of a condition
  // subsequent indented lines for specified actions
#EndIf
```

The #EndIf directive keyword should appear on a line by itself. Anything following #EndIf on the same line is considered a comment and is not parsed.

For an example, see **#If**.

9.2.8 #Execute

The **#Execute** preprocessor directive executes a line of ObjectScript at compile time. It has the form:

```
#Execute <ObjectScript code>
```

where the content that follows **#Execute** is valid ObjectScript code. This code can refer to any variable or property that has a value at compile time; it can also invoke any method or routine that is available at compile time. ObjectScript commands and functions are always available to be invoked.

#Execute does not return any value indicating if the code has successfully run or not. Application code is responsible for checking a status code or other information of this kind; this can use additional **#Execute** directives or other code.

Note: There may be unexpected results if you use **#Execute** with local variables. Reasons for this include:

- A variable used at compile time may be out of scope at runtime.
- With multiple routines or methods, the variable may not be available when referenced. This issue may be exacerbated by the fact that the application programmer does not control compilation order.

For example, you can determine the day of the week at compile time and save it using the following code:

```
#Execute KILL ^DayOfWeek
#Execute SET ^DayOfWeek = $ZDate($H,12)

 WRITE "Today is ",^DayOfWeek,".",!
```

where the ^*DayOfWeek* global is updated each time compilation occurs.

9.2.9 #If

The **#If** preprocessor directive begins a block of conditional text. It takes an ObjectScript expression as an argument, tests the truth value of the argument, and compiles a block of code if the truth value of its argument is true. The block of code concludes with a **#Else**, **#ElseIf**, or **#EndIf** directive.

```
#If <expression>
  // subsequent indented lines for specified actions

  // next preprocessor directive
```

where <*expression*> is a valid ObjectScript expression. If <*expression*> evaluates to a non-zero value, it is true.

For example:

```
 KILL ^MyColor, ^MyNumber
#Define ColorDay $ZDate($H,12)
#If $$$ColorDay="Monday"
 SET ^MyColor = "Red"
 SET ^MyNumber = 1
#ElseIf $$$ColorDay="Tuesday"
 SET ^MyColor = "Orange"
 SET ^MyNumber = 2
#ElseIf $$$ColorDay="Wednesday"
 SET ^MyColor = "Yellow"
 SET ^MyNumber = 3
#ElseIf $$$ColorDay="Thursday"
 SET ^MyColor = "Green"
 SET ^MyNumber = 4
#ElseIf $$$ColorDay="Friday"
 SET ^MyColor = "Blue"
 SET ^MyNumber = 5
#Else
 SET ^MyColor = "Purple"
 SET ^MyNumber = -1
#EndIf
 WRITE ^MyColor, ", ", ^MyNumber
```

This code sets the value of the `ColorDay` macro to the name of the day at compile time. The conditional statement that begins with **#If** then uses the value of `ColorDay` to determine how to set the value of the ^*MyColor* variable. This code has multiple conditions that can apply to `ColorDay` — one for each weekday after Monday; the code uses the **#ElseIf** directive to check these. The fall-through case is the code that follow the **#Else** directive. The **#EndIf** closes the conditional.

Any number of spaces may separate #If and <*expression*>. However, no spaces are permitted within <*expression*>. Anything following <*expression*> on the same line is considered a comment and is not parsed.

Note: If **#If** appears in method code and has an argument other than a literal value of 0 or 1, the compiler generates code in subclasses (rather than invoking the method in the superclass). To avoid generating this code, test conditions for a value of 0 or 1, which results in simpler code and optimizes performance.

9.2.10 #IfDef

The **#IfDef** preprocessor directive marks the beginning of a block of conditional code where execution depends on a macro having been defined. It has the form:

```
#IfDef macro-name
```

where *macro-name* appears without any leading "$$$" characters. Anything following *macro-name* on the same line is considered a comment and is not parsed.

Execution of the code is contingent on the macro having been defined. Execution continues until reaching a **#Else** directive or a closing **#EndIf** directive.

#IfDef checks only if a macro has been defined, not what its value is. Hence, if a macro exists and has a value of 0 (zero), **#IfDef** still executes the conditional code (since the macro does exist).

Also, since **#IfDef** checks only the existence of a macro, there is only one alternate case (if the macro is not defined), which the **#Else** directive handles. The **#ElseIf** directive is not for use with **#IfDef**.

For example, the following provides a simple binary switch based on a macro's existence:

```
#Define Heads

#IfDef Heads
  WRITE "The coin landed heads up.",!
#Else
  WRITE "The coin landed tails up.",!
#EndIf
```

9.2.11 #IfNDef

The **#IfNDef** preprocessor directive marks the beginning of a block of conditional code where execution depends on a macro having not been defined. It has the form:

```
#IfNDef macro-name
```

where *macro-name* appears without any leading "$$$" characters. Anything following *macro-name* on the same line is considered a comment and is not parsed.

Execution of the code is contingent on the macro having *not* been defined. Execution continues until reaching a **#Else** directive or a closing **#EndIf** directive. The **#ElseIf** directive is not for use with **#IfNDef**.

Note: **#IfNDef** has an alternate name of **#IfUnDef**. The two names behave identically.

For example, the following provides a simple binary switch based on a macro *not* having been defined:

```
#Define Multicolor 256

#IfNDef Multicolor
  SET NumberOfColors = 2
#Else
  SET NumberOfColors = $$$Multicolor
#EndIf
  WRITE "There are ",NumberOfColors," colors in use.",!
```

9.2.12 #Import

The **#Import** preprocessor directive specifies the schema search path for any subsequent Embedded SQL DML statements.

#Import specifies one or more schema names to search to supply the schema name for an unqualified table, view, or stored procedure name. You can specify a single schema name, or a comma-separated list of schema names. Schemas are searched in the current namespace. This is shown in the following example, which locates the Employees.Person table:

```
#Import Customers,Employees,Sales
  &sql(SELECT Name,DOB INTO :n,:date FROM Person)
  WRITE "name: ",n," birthdate: ",date,!
  WRITE "SQLCODE=",SQLCODE
```

All of the schemas specified in the **#Import** directive are searched. The Person table must be found in exactly one of the schemas listed in **#Import**. Because **#Import** requires a match within the schema search path, the system-wide default schema is not used.

Dynamic SQL uses the %SchemaPath property to supply a schema search path to resolve unqualified names.

Both **#Import** and #SQLCompile Path specify one or more schema names used to resolve an unqualified table name. Some of the differences between these two directives are as follows:

- **#Import** detects ambiguous table names. **#Import** searches all specified schemas, detecting all matches. #SQLCompile Path searches the specified list of schemas in left-to-right order until it finds the first match. Therefore, **#Import** can detect ambiguous table names; **#SQLCompile Path** cannot. For example, **#Import Customers,Employees,Sales** must find exactly one occurrence of Person in the Customers, Employees, and Sales schemas; if it finds more than one occurrence of this table name an SQLCODE -43 error occurs: "Table 'PERSON' is ambiguous within schemas".
- **#Import** cannot take the system-wide default. If **#Import** cannot find the Person table in any of its listed schemas, an SQLCODE -30 error occurs. If **#SQLCompile Path** cannot find the Person table in any of its listed schemas, it checks the system-wide default schema.
- **#Import** directives are additive. If there are multiple **#Import** directives, the schemas in all of the directives must resolve to exactly one match. Specifying a second **#Import** does not inactivate the list of schema names specified in

a prior **#Import**. Specifying an **#SQLCompile Path** directive overwrites the path specified in a prior **#SQLCompile Path** directive; **#SQLCompile Path** does not overwrite schema names specified in prior **#Import** directives.

InterSystems IRIS ignores non-existent schema names in **#Import** directives. InterSystems IRIS ignores duplicate schema names in **#Import** directives.

If the table name is already qualified, the **#Import** directives do not apply. For example:

```
#Import Voters
#Import Bloggers
&sql(SELECT Name,DOB INTO :n,:date FROM Sample.Person)
WRITE "name: ",n," birthdate: ",date,!
WRITE "SQLCODE=",SQLCODE
```

In this case, InterSystems IRIS searches the Sample schema for the Person table. It does not search the Voters or Bloggers schemas.

- **#Import** is applied to SQL DML statements. It can be used to resolve unqualified table names and view names for SQL **SELECT** queries, and for **INSERT**, **UPDATE**, and **DELETE** operations. **#Import** can also be used to resolve unqualified procedure names in SQL CALL statements.
- **#Import** is not applied to SQL DDL statements. It cannot be used to resolve unqualified table, view, and procedure names in data definition statements such as **CREATE TABLE** and the other CREATE, ALTER, and DROP statements. If you specify an unqualified name for a table, view, or stored procedure when creating, modifying, or deleting the definition of this item, InterSystems IRIS will ignore **#Import** values and use the system-wide default schema.

Compare with the #SQLCompile Path preprocessor directive.

9.2.13 #Include

The **#Include** preprocessor directive loads a specified file name that contains preprocessor directives. It has the form:

```
#Include <filename>
```

where *filename* is the name of the include file, not including the `.inc` suffix. Include files are typically located in the same directory as the file calling them. Their names are case-sensitive.

To list all of the system-supplied **#Include** *filename*s, issue the following command:

```
ZWRITE ^rINC("%occInclude",0)
```

To list the contents of one of these **#Include** files, specify the desired include file. For example:

```
ZWRITE ^rINC("%occStatus",0)
```

To list the **#Include** files pre-processed when generating an INT routine, use the ^ROUTINE global. Note that these **#Include** directives do not have to be referenced in the ObjectScript code:

```
ZWRITE ^ROUTINE("myroutine",0,"INC")
```

Note: When using **#Include** in stored procedure code, it must be preceded by a colon character ":", such as:

```
CREATE PROCEDURE SPxx() Language OBJECTSCRIPT {
 :#Include %occConstant
     SET x=##Lit($$$NULLOREF)
}
```

When including files at the beginning of a class, the directive does not include the pound sign. Hence, for a single file, it is:

```
Include MyMacros
```

For multiple files, it is:

```
Include (MyMacros, YourMacros)
```

For example, suppose there is an OS.inc header file that contains macros:

```
 #Define Windows
 #Define UNIX
```

```
#Include OS

#IfDef Windows
  WRITE "The operating system is not case-sensitive.",!
#Else
  WRITE "The operating system is case-sensitive.",!
#EndIf
```

9.2.14 #NoShow

The **#NoShow** preprocessor directive ends a comment section that is part of an include file. It has the form:

```
#NoShow
```

where **#NoShow** follows a **#Show** directive. It is strongly recommended that every **#Show** have a corresponding **#NoShow**, even when the comment section continues to the end of the file. For an example, see the entry for **#Show**.

9.2.15 #Show

The **#Show** preprocessor directive begins a comment section that is part of an include file. By default, comments in an include file do not appear within the calling code. Hence, include file comments outside the **#Show**—**#NoShow** bracket do not appear in the referencing code.

The directive has the form:

```
#Show
```

It is strongly recommended that every **#Show** have a corresponding **#NoShow**, even when the comment section continues to the end of the file.

In the following example, the file OS.inc (from the **#Include** example) includes the following comments:

```
#Show
  // If compilation fails, check the file
  // OS-errors.log for the statement "No valid OS."
#NoShow
  // Valid values for the operating system are
  // Windows or UNIX (and are case-sensitive).
```

where the first two lines of comments (starting with "If compilation fails...") appear in the code that includes the include file and the second two lines of comments (starting with "Valid values...") appear only in the include file itself.

9.2.16 #SQLCompile Audit

The **#SQLCompile Audit** preprocessor directive is a boolean that specifies whether any subsequent Embedded SQL statements should be audited. It has the form:

```
#SQLCompile Audit=value
```

where *value* is either ON or OFF.

For this macro preprocessor directive to have any effect, the %System/%SQL/EmbeddedStatement system audit event must be enabled. By default, this system audit event is not enabled.

9.2.17 #SQLCompile Mode

The **#SQLCompile Mode** preprocessor directive specifies the compilation mode for any subsequent Embedded SQL statements. It has the form:

```
#SQLCompile Mode=value
```

where *value* is one of the following:

- Embedded — Compiles ObjectScript code and embedded SQL code prior to runtime. This is the default.
- Deferred — Compiles ObjectScript code, but defers compiling embedded SQL code until runtime. This enables you to compile a routine containing SQL that references a table that does not yet exist at compile time.

  Note: `#SQLCompile Mode=Deferred` should not be confused with the similarly-name **%SYSTEM.SQL.SetCompileModeDeferred()** method, which is used for a completely different purpose.

Deferred mode and Embedded mode statements are otherwise identical. Both Deferred mode statements and Embedded mode statements are static. That is, they cannot be dynamically assembled at runtime and submitted for processing. If dynamic code is required, use Dynamic SQL, as described in *Using InterSystems SQL*.

Deferred mode can be used for **INSERT**, **UPDATE**, and **DELETE** operations, and **SELECT** statements that return a single row of data. Deferred SQL cannot be used for multi-row **SELECT** statements that declare a cursor and fetch rows of data; attempting to do so generates a #5663 compilation error. Like Embedded SQL, Deferred SQL does not check privileges.

In Deferred mode, SQL can refer to tables, user-defined functions, and other entities that do not yet exist at compile time.

If an embedded SQL statement contains an invalid SQL statement (for example, an SQL syntax error), the Macro Preprocessor generates the code "** SQL Statement Failed to Compile **" and continues to compile ObjectScript code. Thus when compiling a class with a method that contains invalid embedded SQL, the SQL error is reported, but the method is generated. The invalid SQL causes an error when this method is run.

Embedded SQL provides optimal SQL performance, and should be used whenever possible. Deferred SQL is generally more efficient than Dynamic SQL. Deferred SQL may be needed when converting Transact-SQL or Informix SPL stored procedures to InterSystems IRIS. The provided conversion tools for stored procedures support this feature.

For further details, refer to the Embedded SQL chapter of *Using InterSystems SQL*.

9.2.18 #SQLCompile Path

The **#SQLCompile Path** preprocessor directive specifies the schema search path for any subsequent Embedded SQL DML statements. It has the form:

```
#SQLCompile Path=schema1[,schema2[,...]]
```

where *schema* is a schema name used to look up an unqualified SQL table name, view name, or procedure name in the current namespace. You can specify one schema name or a comma-separated list of schema names. Schemas are searched in the order specified. Searching ends and the DML operation is performed when the first match occurs. If none of the schemas contain a match, the system-wide default schema is searched.

Because schemas are searched in the specified order, there is no detection of ambiguous table names. The #Import preprocessor directive also supplies a schema name to an unqualified SQL table, view, or procedure name from a list of schema names; **#Import** does detect ambiguous names.

InterSystems IRIS ignores non-existent schema names in **#SQLCompile Path** directives. InterSystems IRIS ignores duplicate schema names in **#SQLCompile Path** directives.

- **#SQLCompile Path** is applied to SQL DML statements. It can be used to resolve unqualified table names and view names for SQL **SELECT** queries, and for **INSERT**, **UPDATE**, and **DELETE** operations. **#SQLCompile Path** can also be used to resolve unqualified procedure names in SQL CALL statements.
- **#SQLCompile Path** is not applied to SQL DDL statements. It cannot be used to resolve unqualified table, view, and procedure names in data definition statements such as **CREATE TABLE** and the other CREATE, ALTER, and DROP statements. If you specify an unqualified name for a table, view, or stored procedure when creating, modifying, or deleting the definition of this item, InterSystems IRIS will ignore **#SQLCompile Path** values and use the system-wide default schema.

Dynamic SQL uses the %SchemaPath property to supply a schema search path to resolve unqualified names.

The following example resolves the unqualified table name Person to the Sample.Person table. It first searches the Cinema schema (which does not contain a table named Person), then searches the Sample schema:

```
#SQLCompile Path=Cinema,Sample
  &sql(SELECT Name,Age
       INTO :a,:b
       FROM Person)
  WRITE "Name is: ",a,!
  WRITE "Age is: ",b
```

In addition to specifying schema names as search path items, you can specify the following keywords:

- CURRENT_PATH: specifies the current schema search path, as defined in a prior **#SQLCompile Path** preprocessor directive. This is commonly used to add schemas to the beginning or end of an existing schema search path, as shown in the following example:

  ```
  #SQLCompile Path=schema_A,schema_B,schema_C
  #SQLCompile Path=CURRENT_PATH,schema_D
  ```

- CURRENT_SCHEMA: specifies the current schema container class name. If **#SQLCompile Path** is defined in a class method, the CURRENT_SCHEMA is the schema mapped to the current class package. If **#SQLCompile Path** is defined in a .MAC routine, the CURRENT_SCHEMA is the configuration default schema.

  For example, if you define a class method in the class User.MyClass that specifies **#SQLCompile Path=CURRENT_SCHEMA**, the CURRENT_SCHEMA will (by default) resolve to SQLUser, since SQLUser is the default schema name for the User package. This is useful when you have a superclass and subclass in different packages, and you define a method in the superclass that has an SQL query with an unqualified table name. Using CURRENT_SCHEMA, you can have the table name resolve to the superclass schema in the superclass and to the subclass schema in the subclass. Without the CURRENT_SCHEMA search path setting, the table name would resolve to the superclass schema in both classes.

  If **#SQLCompile Path=CURRENT_SCHEMA** is used in a trigger, the schema container class name is used. For example, if class pkg1.myclass has a trigger than specifies **#SQLCompile Path=CURRENT_SCHEMA**, and class pkg2.myclass extends pkg1.myclass, InterSystems IRIS resolves the non-qualified table names in the SQL statements in the trigger to the schema for package pkg2 when the pkg2.myclass class is compiled.

- DEFAULT_SCHEMA specifies the system-wide default schema. This keyword enables you to search the system-wide default schema as a item within the schema search path, before searching other listed schemas. The system-wide default schema is always searched after searching the schema search path if all the schemas specified in the path have been searched without a match.

If you specify a schema search path, the SQL query processor uses the schema search path first when attempting to resolve an unqualified name. If it does not find the specified table or procedure, it then looks in the schema(s) that are provided via **#Import** (if specified), or the configured system-wide default schema. If it does not find the specified table in any of these places, it generates an SQLCODE -30 error.

#SQLCompile Path can be used with **#SQLCompile Mode** values Embedded or Deferred. For further details, refer to the Embedded SQL chapter of *Using InterSystems SQL*.

The scope of the schema search path is the routine or method it is defined in. If a schema path is specified in a class method, it only applies to that class method, and not to other methods in the class. If it is specified in a .MAC routine, it applies from that point forward in the routine until another **#SQLCompile Path** directive is found, or the end of the routine is reached.

Schemas are defined for the current namespace.

Compare with the #Import preprocessor directive.

9.2.19 #SQLCompile Select

The **#SQLCompile Select** preprocessor directive specifies the data format mode for any subsequent Embedded SQL statements. It has the form:

```
#SQLCompile Select=value
```

where *value* is one of the following:

- Display — Formats data for screen and print.
- Logical — Leaves data in its in-memory format.
- ODBC — Formats data for presentation via ODBC or JDBC.
- Runtime — Supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format based on the execution-time select mode value. The output values are converted to the current mode.

  You can get the execution-time select mode value using the **GetSelectMode** method of the %SYSTEM.SQL class. You can set the execution-time select mode value using the **SetSelectMode** method of the %SYSTEM.SQL class.
- Text — Synonym for Display.
- FDBMS — Allows Embedded SQL to format data the same as FDBMS.

The value of this macro determines the Embedded SQL output data format for **SELECT** output host variables, and the required input data format for Embedded SQL **INSERT**, **UPDATE**, and **SELECT** input host variables. For details, refer to The Macro Preprocessor in the "Using Embedded SQL" chapter of *Using InterSystems SQL*.

The following Embedded SQL examples use the different compile modes to return three fields from the Sample.Person table, which are Name (a string field), DOB (a date field), and Home (a list field):

```
#SQLCOMPILE SELECT=Logical
  &sql(SELECT Name,DOB,Home
       INTO :n,:d,:h
       FROM Sample.Person)
  WRITE "name is: ",n,!
  WRITE "birthdate is: ",d,!
  WRITE "home is: ",h
```

```
#SQLCOMPILE SELECT=Display
  &sql(SELECT Name,DOB,Home
       INTO :n,:d,:h
       FROM Sample.Person)
  WRITE "name is: ",n,!
  WRITE "birthdate is: ",d,!
  WRITE "home is: ",h

#SQLCOMPILE SELECT=ODBC
  &sql(SELECT Name,DOB,Home
       INTO :n,:d,:h
       FROM Sample.Person)
  WRITE "name is: ",n,!
  WRITE "birthdate is: ",d,!
  WRITE "home is: ",h

#SQLCOMPILE SELECT=Runtime
  &sql(SELECT Name,DOB,Home
       INTO :n,:d,:h
       FROM Sample.Person)
  WRITE "name is: ",n,!
  WRITE "birthdate is: ",d,!
  WRITE "home is: ",h
```

9.2.20 #UnDef

The **#UnDef** preprocessor directive removes the definition for an already-defined macro. It has the form:

```
#UnDef macro-name
```

where *macro-name* is a macro that has already been defined.

#UnDef follows an invocation of **#Define** or **#Def1Arg**. It works in conjunction with **#IfDef** and its associated preprocessor directives (**#Else**, **#EndIf**, and **#IfNDef**).

The following example demonstrates code that is conditional on a macro being defined and then undefined.

```
#Define TheSpecialPart

#IfDef TheSpecialPart
  WRITE "We're in the special part of the program.",!
#EndIf

 //
 // code here...
 //

#UnDef TheSpecialPart

#IfDef TheSpecialPart
  WRITE "We're in the special part of the program.",!
#Else
  WRITE "We're no longer in the special part of the program.",!
#EndIf

#IfNDef TheSpecialPart
  WRITE "We're still outside the special part of the program.",!
#Else
  WRITE "We're back inside the special part of the program.",!
#EndIf
```

where the .int code for this is:

```
  WRITE "We're in the special part of the program.",!
  //
  // code here...
  //
  WRITE "We're no longer in the special part of the program.",!
  WRITE "We're still outside the special part of the program.",!
```

9.2.21 ##;

The **##;** preprocessor directive makes the remaining part of the current line a comment that does not appear in .int code. The comment appears only in either .mac code or in an include file. The **##;** comment indicator should always be used for comments in a preprocessor directive:

```
#Define alphalen ##Function($LENGTH("abcdefghijklmnopqrstuvwxyz")) ##; + 100
   WRITE $$$alphalen," is the length of the alphabet"
```

A **##;** comment indicator can appear in a #Define, #Def1Arg, or #Dim preprocessor directive. It cannot be used following a ##Continue preprocessor directive. Use of // or ; remainder-of-the-line comments should be avoided in preprocessor directives.

##; may also be used anywhere in an ObjectScript code line or an Embedded SQL code line to specify a comment that does not appear in .int code. The comment continues for the remainder of the current line.

##; is evaluated before evaluation of Embedded HTML or Embedded JavaScript.

Compare with #;, which appears in column 1 and makes an entire line a comment. **##;** makes the rest of the current line a comment. When **##;** appears in the first column of the line, it is functionally identical to the #; preprocessor directive.

9.2.22 ##Continue

The **##Continue** preprocessor directive continues a macro definition on the next line, to support multiline macro definitions. It appears at the end of a line of a macro definition to signal the continuation, in the form:

```
#Define <beginning of macro definition> ##Continue
     <continuation of macro definition>
```

A macro definition can use multiple **##Continue** directives.

For example,

```
#Define Multiline(%a,%b,%c) ##Continue
    SET v=" of Oz" ##Continue
    SET line1="%a"_v ##Continue
    SET line2="%b"_v ##Continue
    SET line3="%c"_v

 $$$Multiline(Scarecrow,Tin Woodman,Lion)
 WRITE "Here is line 1: ",line1,!
 WRITE "Here is line 2: ",line2,!
 WRITE "Here is line 3: ",line3,!
```

##Continue must appear at the end of a macro definition line. Therefore, **##Continue** cannot be followed by a ##; comment or a /* comment text */ comment. The #; full-line comment also cannot be used within a **##Continue** multiline directive. You can comment a **##Continue** line as follows:

```
#Define Multiline(%a,%b,%c) ##Continue
    SET v=" of Oz" /* set a variable to a string */ ##Continue
    SET line1="%a"_v ##Continue
    SET line2="%b"_v ##Continue
    SET line3="%c"_v
```

9.2.23 ##Expression

The **##Expression** preprocessor function evaluates an ObjectScript expression at compile time. It has the form:

```
##Expression(content)
```

where *content* is valid ObjectScript code that does not include any quoted strings or any preprocessor directives (with the exception of a nested **##Expression**, as described below).

The preprocessor evaluates the value of the function's argument at compile time and replaces ##Expression(content) with the evaluation in the ObjectScript .int code. Variables must appear in quotation marks within **##Expression**; otherwise, they are evaluated at compile time.

The following example shows some simple expressions:

```
#Define NumFunc ##Expression(1+2*3)
#Define StringFunc ##Expression("""This is"_" a concatenated string""")
  WRITE $$$NumFunc,!
  WRITE $$$StringFunc,!
```

The following example defines an expression containing the compile timestamp of the current routine:

```
#Define CompTS ##Expression("""Compiled: " _ $ZDATETIME($HOROLOG) _ """,!")
  WRITE $$$CompTS
```

where the argument of **##Expression** is parsed in three parts, which are concatenated using the "_" operator:

- The initial string, `"""Compiled: "`. This is delimited by double-quotes. Within that, the pair of double-quotes specifies a double-quote to appear *after* evaluation.
- The value, `$ZDATETIME($HOROLOG)`. The value of the **$HOROLOG** special variable at compile-time, as converted and formatted by the **$ZDATETIME** function.
- The final string, `""",!"`. This is also delimited by double-quotes. Within that, there are a pair of double-quotes (which results in a single double-quote after evaluation). Since the value being defined is being passed to the **WRITE** command, the final string includes `,!`, so that the **WRITE** command includes a carriage return.

The routine's intermediate (.int) code would then include a line such as:

```
  WRITE "Compiled: 05/29/2018 07:49:30",!
```

9.2.23.1 ##Expression and Literal Strings

Parsing with **##Expression** does not recognize literal strings; bracketing characters inside of quotes are not treated specially. For example, in the directive:

```
#Define MyMacro ##Expression(^abc(")",1))
```

the quoted right parenthesis is treated as if it is a closing parenthesis for specifying the argument.

9.2.23.2 ##Expression Nesting

InterSystems IRIS supports nested **##Expression**s. You can define an **##Expression** that contains macros that expand to other **##Expression**s, as long as the expansion can be evaluated at the ObjectScript level (that is, it contains no preprocessor directives) and stored in an ObjectScript variable. With nested **##Expression**s, the macros with the **##Expression** expression are expanded first, then the nested **##Expression** is expanded.

##Expression can also nest the following macro functions: **##BeginLit**...**##EndLit**, ##Function, ##Lit, ##Quote, ##SafeExpression, ##StripQ, ##Unique.

9.2.23.3 ##Expression, Subclasses, and ##SafeExpression

When a method contains an **##Expression** this is detected when the class is compiled. Because the compiler does not parse the content of the **##Expression**, this **##Expression** could generate different code in a subclass. To avoid this, InterSystems IRIS causes the compiler to regenerate the method code for each subclass. For example, `##Expression(%classname)` inserts the current classname; when you compile a subclass, the code expects it will insert the subclass classname. InterSystems IRIS forces this method to be regenerated in the subclass to ensure that this occurs.

If you know that the code will never be different in a subclass, you can avoid regenerating the method for each subclass. To do this, substitute the **##SafeExpression** preprocessor function for **##Expression**. These two preprocessor functions are otherwise identical.

9.2.23.4 How ##Expression Works

The argument to **##Expression** is set into a value via the ObjectScript XECUTE command:

```
 SET value="Set value="_expression XECUTE value
```

where *expression* is an ObjectScript expression that determines the value of *value* and may not contain macros or a **##Expression** preprocessor function.

However, the results of the `XECUTE value` may contain macros, another **##Expression**, or both. The ObjectScript preprocessor further expands any of these, as in this example.

Suppose the content of routine A.mac includes:

```
#Define BB ##Expression(10_"_"_$$aa^B())
 SET CC = $$$BB
 QUIT
```

and routine B.mac includes:

```
aa()
 QUIT "##Expression(10+10+10)"
```

A.int then includes the following:

```
 SET CC = 10_30
 QUIT
```

9.2.24 ##Function

The **##Function** preprocessor function evaluates an ObjectScript function at compile time. It has the form

```
##Function(content)
```

where *content* is an ObjectScript function and can be user-defined. **##Function** replaces `##Function(content)` with the returned value from the function.

The following example returns the value from an ObjectScript function:

```
#Define alphalen ##Function($LENGTH("abcdefghijklmnopqrstuvwxyz"))
   WRITE $$$alphalen
```

In the following example, suppose there is a user-defined function in the GetCurrentTime.mac file:

```
Tag1()
 KILL ^x
 SET ^x = """" _ $Horolog _ """"
 QUIT ^x
```

It is then possible to invoke this code in a separate routine, called ShowTimeStamps.mac, as follows:

```
Tag2
#Define CompiletimeTimeStamp ##function($$Tag1^GetCurrentTime())
#Define RuntimeTimeStamp $$Tag1^GetCurrentTime()
 SET x=$$$CompiletimeTimeStamp
 WRITE x,!
 SET y=$$$RuntimeTimeStamp
 WRITE y,!
```

The output of this at the Terminal is something like:

```
USER>d ^ShowTimeStamps
64797,43570
"64797,53807"

USER>
```

where the first line of output is the value of **$Horolog** at compile time and the second line is the value of **$Horolog** at runtime. (The first line of output is not quoted and the second line is quoted because *x* substitutes a quoted string for its value, so there are no quotes displayed in the Terminal, while *y* prints the quoted string directly to the Terminal.)

Note: It is the responsibility of the application programmer to make sure that the return value of the **##Function** call makes both semantic and syntactic sense, given the context of the call.

9.2.24.1 ##Function Nesting

InterSystems IRIS supports nested within a **##Function**. **##Function** can nest the following macro functions: **##BeginLit**...**##EndLit**, **##Function**, ##Lit, ##Quote, ##Expression, ##SafeExpression, ##StripQ, and ##Unique.

9.2.25 ##Lit

The **##Lit** preprocessor function preserves the content of its argument in literal form:

```
##Lit(content)
```

where *content* is a string that is valid ObjectScript expression. The **##Lit** preprocessor function ensures that the string it receives is not evaluated, but that it is treated as literal text.

For example, the following code:

```
 #Define Macro1 "Row 1 Value"
 #Define Macro2 "Row 2 Value"
   ##Lit(;;) Column 1 Header ##Lit(;) Column 2 Header
   ##Lit(;;) Row 1 Column 1  ##Lit(;) $$$Macro1
   ##Lit(;;) Row 2 Column 1  ##Lit(;) $$$Macro2
```

creates a set of lines that form a table in .int code:

```
 ;; Column 1 Header ; Column 2 Header
 ;; Row 1 Column 1  ; "Row 1 Value"
 ;; Row 2 Column 1  ; "Row 2 Value"
```

By using the **##Lit** preprocessor function, macros are evaluated and are delimited by the semicolons in the .int code

9.2.26 ##Quote

The **##Quote** preprocessor function takes a single argument and returns that argument quoted. If the argument already contains quote characters it escapes these quote characters by doubling them. It has the form:

```
##Quote(value)
```

where *value* is a literal that is converted to a quoted string. In *value* a parenthesis character or a quote character must be paired. For example, the following is a valid *value*:

```
#Define qtest ##Quote(He said "Yes" after much debate)
   ZZWRITE $$$qtest
```

it returns `"He said ""Yes"" after much debate"`. `##Quote(This (") is a quote character)` is not a valid *value*.

Parentheses within a *value* string must be paired. The following is a valid *value*:

```
#Define qtest2 ##Quote(After (a lot of) debate)
   ZZWRITE $$$qtest2
```

The following example shows the use of **##Quote**:

```
#Define AssertEquals(%e1,%e2) DO AssertEquals(%e1,%e2,##Quote(%e1)_" == "_##Quote(%e2))
Main ;
  SET a="abstract"
  WRITE "Test 1:",!
  $$$AssertEquals(a,"abstract")
  WRITE "Test 2:",!
  $$$AssertEquals(a_"","abstract")
  WRITE "Test 3:",!
  $$$AssertEquals("abstract","abstract")
  WRITE "All done"
  QUIT
AssertEquals(e1,e2,desc) ;
  WRITE desc_" is "_$SELECT(e1=e2:"true",1:"false"),!
  QUIT
```

9.2.26.1 ##Quote Nesting

InterSystems IRIS supports nested within a **##Quote** function. **##Quote** can nest the following macro functions: **##BeginLit**...**##EndLit**, ##Function, ##Lit, **##Quote**, ##Expression, ##SafeExpression, ##StripQ, and ##Unique.

9.2.27 ##QuoteExp

The **##QuoteExp** preprocessor function takes as an argument an expression that gets evaluated during compilation. This expression can contain nested/recursive MPP functions. It then returns the complied result as a quoted string. If the argument already contains quote characters it escapes these quote characters by doubling them. It has the form:

```
##QuoteExp(expression)
```

where *expression* may contain any of the following nested/recursive MPP functions: **##BeginLit**...**##EndLit**, ##Expression, ##Function, ##Lit, ##Quote, **##QuoteExp**, ##SafeExpression, ##StripQ, ##Unique, and **##This**.

By using **##QuoteExp** you can create a general-purpose complex global macro that accepts a variable number of subscripts and returns that reference as a quoted string, whether the subscript values are passed as numeric or string. You define a macro as a complex global reference using #Def1Arg directive. To return this complex global reference as a quoted string, regardless of the subscripts provided to the macro, wrapper this macro in **##QuoteExp**. The macro evaluates the expression argument passed to **##QuoteExp** and returns this value as a quoted string.

For example:

```
#Def1Arg complexGlobal(%subs)
^GLO("dd"##expression($s(%literalargs'=$lb(""):","_$LTS(%literalargs,","),1:"")))
#Def1Arg complexGlobalQE(%subs) ##QuoteExp($$$complexGlobal(%subs))
```

9.2.28 ##SQL

The **##SQL** preprocessor directive invokes a specified SQL statement at compile time. It has the form:

```
##SQL(SQL-statement)
```

where *SQL-statement* is a valid SQL statement. The **##SQL** preprocessor directive is analogous to the **&SQL** directive — **##SQL()** invokes the statement at compile time, which **&SQL()** does so at runtime.

If a **##SQL** directive contains invalid SQL (such as a syntax error) or refers to a nonexistent table or column, then the macro preprocessor generates a compilation error.

For example, the following code runs a query at first at compile time and then again at runtime:

```
 ##sql(SELECT COUNT(*) INTO :count1 FROM Sample.Person)

 &sql(SELECT COUNT(*) INTO :count2 FROM Sample.Person)

 WRITE "Number of instances of Sample.Person at compile time: ",count1,!
 WRITE "Number of instances of Sample.Person at runtime:      ",count2,!
```

9.2.29 ##StripQ

The **##StripQ** preprocessor function takes a single argument and returns that argument with quotes removed. It is the inverse of the ##Quote macro function.

```
##StripQ(value)
```

where *value* is a literal or variable from which enclosing quotes, if present, are stripped.

9.2.30 ##Unique

The **##Unique** preprocessor function creates a new, unique local variable within a macro definition for use at compile time or runtime. This preprocessor function is available for use only as part of **#Define** or **#Def1Arg** call. It has the form:

```
##Unique(new)
##Unique(old)
```

where `new` specifies the creation of a new, unique variable and `old` specifies a reference to that same variable.

The variable created by `SET ##Unique(new)` is a local variable with the name %mmmu1, subsequent `SET ##Unique(new)` operations create local variables with the names %mmmu2, %mmmu3, and so forth. These local variables are subject to the same scoping rules as all % local variables; % variables are always public variables. Like all local variables, they can be displayed using **ZWRITE** and can be killed using an argumentless **KILL**.

User code can refer to the `##Unique(old)` variable just as it can refer to any other ObjectScript variable. The `##Unique(old)` syntax can be used an indefinite number of times to refer to the created variable.

Subsequent calls to `##Unique(new)` create a new variable; after calling `##Unique(new)` again, subsequent calls to `##Unique(old)` refer to the subsequently created variable.

For example, the following code uses `##Unique(new)` and `##Unique(old)` to swap values between two variables:

```
#Define Switch(%a,%b) SET ##Unique(new)=%a, %a=%b, %b=##Unique(old)
READ "First variable value? ",first,!
READ "Second variable value? ",second,!
$$$Switch(first,second)
WRITE "The first value is now ",first," and the second is now ",second,!
```

To maintain uniqueness of these variables:

- Do not attempt to set `##Unique(new)` outside of a **#Define** or **#Def1Arg** preprocessor directive.
- Do not set `##Unique(new)` in a preprocessor directive within a method or procedure. These will generate a variable name that is unique to the method (%mmmu1); however, because this is a % variable, it is globally scoped. Invoking another method that sets `##Unique(new)` also creates %mmmu1, overwriting the variable created by the first method.
- Never set a %mmmu1 variable directly. InterSystems IRIS reserves all % variables (except %z and %Z variables) for system use; they should never be set by user code.

9.3 Using System-supplied Macros

This section describes topics related to some of the predefined macros available with InterSystems IRIS. Its topics are:

- Making System-supplied Macros Accessible
- System-supplied Macro Reference

9.3.1 Making System-supplied Macros Accessible

These macros are available to all subclasses of %RegisteredObject. To make these available within a routine or a class that does not extend %RegisteredObject, include the appropriate file:

- For status-related macros, include %occStatus.inc.
- For message-related macros, include %occMessages.inc

See each macro below for which include file it requires.

The syntax for such statements is:

```
#Include %occStatus
```

The names of these include files are case-sensitive. For more details on using externally defined macros, see the section "Referring to External Macros (Include Files)."

9.3.2 System-supplied Macro Reference

Macro names are case-sensitive. Among the macros supplied with InterSystems IRIS are:

ADDSC(sc1, sc2)

The ADDSC macro appends a %Status code (*sc2*) to an existing %Status code (*sc1*). This macro requires %occStatus.inc.

EMBEDSC(sc1, sc2)

The EMBEDSC macro embeds a %Status code (*sc2*) within an existing %Status code (*sc1*). This macro requires %occStatus.inc.

ERROR(errorcode, arg1, arg2, ...)

The ERROR macro creates a %Status object using an object error code (*errorcode*) the associated text of which may accept some number of arguments of the form %1, %2, and so on. ERROR then replaces these arguments with the macro arguments that follow *errorcode* (*arg1*, *arg2*, and so on) based on the order of these additional arguments. This macro requires %occStatus.inc.

For a list of system-defined error codes, see "General Error Messages" in the *InterSystems IRIS Error Reference*.

FormatMessage(language,domain,id,default,arg1,arg2,...)

The FormatMessage macro enables you to retrieve text from the Message Dictionary, and substitute text for message arguments, all in the same macro call. It returns a %String.

Argument	Description
language	An RFC1766 language code. Within a web application, you can specify `%response.Language` to use the default locale.
domain	The message domain. Within a web application, you may specify `%response.Domain`
id	The message ID.
default	The string to use if the message identified by *language*, *domain*, and *id* is not found.
arg1, *arg2*, and so on	Substitution text for the message arguments. All of these are optional, so you can use $$$FormatMessage even if the message has no arguments.

For information on the Message Dictionary, see "Performing Localization" in *Implementing InterSystems Business Intelligence*.

This macro requires %occMessages.inc.

Also see the **FormatMessage()** instance method of %Library.MessageDictionary.

FormatText(text, arg1, arg2, ...)

The `FormatText` macro accepts an input text message (*text*) which may contain arguments of the form `%1`, `%2`, etc. `FormatText` then replaces these arguments with the macro arguments that follow the *text* argument (*arg1*, *arg2*, and so on) based on the order of these additional arguments. It then returns the resulting string. This macro requires %occMessages.inc.

FormatTextHTML(text, arg1, arg2, ...)

The `FormatTextHTML` macro accepts an input text message (*text*) which may contain arguments of the form `%1`, `%2`, etc. `FormatTextHTML` then replaces these arguments with the macro arguments that follow the *text* argument (*arg1*, *arg2*, and so on) based on the order of these additional arguments; the macro then applies HTML escaping. It then returns the resulting string. This macro requires %occMessages.inc.

FormatTextJS(text, arg1, arg2, ...)

The `FormatTextJS` macro accepts an input text message (*text*) which may contain arguments of the form `%1`, `%2`, etc. `FormatTextJS` then replaces these arguments with the macro arguments that follow the *text* argument (*arg1*, *arg2*, and so on) based on the order of these additional arguments; the macro then applies JavaScript escaping. It then returns the resulting string. This macro requires %occMessages.inc.

GETERRORCODE(sc)

The `GETERRORCODE` macro returns the error code value from the supplied %Status code (*sc*). This macro requires %occStatus.inc.

GETERRORMESSAGE(sc,num)

The `GETERRORMESSAGE` macro returns the portion of the error message value from the supplied %Status code (*sc*) as specified by *num*. For example, *num*=1 returns SQLCODE error number, *num*=2 returns the error message text. This macro requires %occStatus.inc.

ISERR(sc)

The `ISERR` macro returns True if the supplied %Status code (*sc*) is an error code. Otherwise, it returns False. This macro requires %occStatus.inc.

ISOK(sc)

The `ISOK` macro returns True if the supplied %Status code (*sc*) is successful completion. Otherwise, it returns False. This macro requires %occStatus.inc.

OK

The `OK` macro creates a %Status code for successful completion. This macro requires %occStatus.inc.

Text(text, domain, language)

The `Text` macro is used for localization. It generates a new message at compile time and generates code to retrieve the message at runtime. This macro requires %occMessages.inc.

TextHTML(text, domain, language)

The `TextHTML` macro is used for localization. It performs the same processing as the `Text` macro; it then additionally applies HTML escaping. It then returns the resulting string. This macro requires %occMessages.inc.

TextJS(text, domain, language)

The `TextJS` macro is used for localization. It performs the same processing as the `Text` macro; it then additionally applies JavaScript escaping. It then returns the resulting string. This macro requires %occMessages.inc.

ThrowOnError(sc)

The `ThrowOnError` macro evaluates the specified %Status code (*sc*). If *sc* represents an error status, `ThrowOnError` performs a **THROW** operation to throw an exception of type %Exception.StatusException to an exception handler. This macro requires %occStatus.inc. For further details, refer to The TRY-CATCH Mechanism in the "Error Processing" chapter of this guide.

THROWONERROR(sc, expr)

The `THROWONERROR` macro evaluates an expression (*expr*), where the expression's value is assumed to be a %Status code; the macro stores the %Status code in the variable passed as *sc*. If the %Status code is an error, `THROWONERROR` performs a **THROW** operation to throw an exception of type %Exception.StatusException to an exception handler. This macro requires %occStatus.inc.

ThrowSQLCODE(sqlcode,message)

The `ThrowSQLCODE` macro uses the specified SQLCODE and Message to perform a **THROW** operation to throw an exception of type %Exception.SQL to an exception handler. This macro requires %occStatus.inc. For further details, refer to The TRY-CATCH Mechanism in the "Error Processing" chapter of this guide.

ThrowSQLIfError(sqlcode,message)

The `ThrowSQLIfError` macro uses the specified SQLCODE and Message to perform a **THROW** operation to throw an exception of type %Exception.SQL to an exception handler. It throws this exception if SQLCODE < 0 (a negative number, indicating an error). This macro requires %occStatus.inc. For further details, refer to The TRY-CATCH Mechanism in the "Error Processing" chapter of this guide.

ThrowStatus(sc)

The `ThrowStatus` macro uses the specified %Status code (*sc*) to perform a **THROW** operation to throw an exception of type %Exception.StatusException to an exception handler. This macro requires %occStatus.inc. For further details, refer to The TRY-CATCH Mechanism in the "Error Processing" chapter of this guide.

10

Embedded SQL

You can embed SQL within ObjectScript on InterSystems IRIS® data platform.

10.1 Embedded SQL

Embedded SQL allows you to include SQL code within an ObjectScript program. The syntax is `&sql( )`. For example:

```
&sql( SELECT Name INTO :n FROM Sample.Person )
WRITE "name is: ",n
```

For further details, refer to the Using Embedded SQL chapter in *Using InterSystems SQL*.

11

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.

11.1 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 can appear in multiple settings.

11.1.1 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.

11.1.2 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.

11.1.3 Settings for Multidimensional Arrays

Multidimensional arrays can appear in three different settings:

- Any global can be used and thereby transformed into an array. In this case, global ^*y* becomes a node in the array ^*y* when you create global ^*y(1)*.
- Any property can be used and thereby transformed into an array. In this case, property *Object.Get(Prop)* becomes a node in the array *Person.Get(Prop)* when you create property *Person.Get(Prop,1)*.
- Any local variable can be used and thereby transformed into an array. In this case, variable *x* becomes a node in array *x* when you create variable *x(1)*.

11.2 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.

11.3 For More Information

For further information on multidimensional arrays, see Using Globals.

12

String Operations

ObjectScript on InterSystems IRIS® data platform 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
- List-Structure String Operations

12.1 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:

  ```
  WRITE $LENGTH("How long is this?")
  ```

  returns 17, the length of a string. For more details, see the $LENGTH reference page in the *ObjectScript Reference*.

- **$JUSTIFY** returns a right-justified string, padded on the left with spaces (and can also perform operations on numeric values). For example, the code:

  ```
  WRITE "one",!,$JUSTIFY("two",8),!,"three"
  ```

  justifies string “two” within eight characters and returns:

  ```
  one
       two
  three
  ```

  For more details, see the $JUSTIFY reference page in the *ObjectScript Reference*.

- **$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:

  ```
  WRITE $ZCONVERT("cRAZy cAPs","t")
  ```

  returns:

  ```
  CRAZY CAPS
  ```

  For more details, see the $ZCONVERT reference page in the *ObjectScript Reference*.

- The **$FIND** function searches for a substring of a string, and returns the position of the character *following* the substring. For example, the code:

  ```
  WRITE $FIND("Once upon a time...", "upon")
  ```

  returns 10 character position immediately following "upon." For more details, see the $FIND reference page in the *ObjectScript Reference*.

- The **$TRANSLATE** function performs a character-by-character replacement within a string. For example, the code:

  ```
  SET text = "11/04/2008"
  WRITE $TRANSLATE(text,"/","-")
  ```

  replaces the date's slashes with hyphens. For more details, see the $TRANSLATE reference page in the *ObjectScript Reference*.

- 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 code:

  ```
  SET text = "green leaves, brown leaves"
  WRITE text,!
  WRITE $REPLACE(text,"leaves","eyes"),!
  WRITE $REPLACE(text,"leaves","hair",15),!
  WRITE text,!
  ```

  performs two distinct operations. 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 the $REPLACE reference page in the *ObjectScript Reference*.

- The **$EXTRACT** function, which returns a substring from a specified position in a string. For example, the code:

  ```
  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 the $EXTRACT reference page in the *ObjectScript Reference*.

12.1.1 Advanced Features of $EXTRACT

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

```
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:

```
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”.

12.2 Delimited Strings

InterSystems IRIS includes functionality that allows you to work with strings as a set of substrings. This functionality provides for the manipulation of related pieces of data that you wish to store as a single whole. These 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:

```
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:

```
WRITE $PIECE("Kennedy Johnson Nixon"," ",2)
```

which returns “Johnson”.

You can also return multiple members of the list, so that the following:

```
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:

```
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.

12.2.1 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,

```
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:

```
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 the $PIECE reference page in the *ObjectScript Reference*.

12.3 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:

```
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*.

12.3.1 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:

```
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:

```
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.)

12.4 Lists and Delimited Strings Compared

12.4.1 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**.

12.4.2 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.

13

Lock Management

A process can apply (lock) and release (unlock) locks using the LOCK command. A lock controls access to a data resource, such as a global variable. This access control is by convention; a lock and its corresponding variable may (and commonly do) have the same name, but are independent of each other. Changing a lock does not affect the variable with the same name; changing a variable does not affect the lock with the same name.

By itself a lock does not prevent another process from modifying the associated data because InterSystems IRIS® data platform does not enforce unilateral locking. Locking works only by convention: it requires that mutually competing processes all implement locking on the same variables.

A lock can be a local (accessible only by the current process) or a global (accessible by all processes). Lock naming conventions are the same as local variable and global variable naming conventions.

A lock remains in effect until it is unlocked by the process that locked it, is unlocked by a system administrator, or is automatically unlocked when the process terminates.

This chapter describes the following topics:

- Management Portal Lock Table, which displays all held locks system-wide, and all lock requests waiting for the release of a held lock. The lock table can also be used to release held locks.
- ^LOCKTAB utility, which returns the same information as the Lock Table.
- Waiting lock requests. How InterSystems IRIS queues lock requests waiting for the release of a held lock.
- Avoiding deadlock (mutually blocking lock requests).

For further information on developing a locking strategy, refer to the article *Locking and Concurrency Control*.

13.1 Managing Current Locks System-wide

InterSystems IRIS maintains a system-wide lock table that records all locks that are in effect and the processes that have locked them, and all waiting lock requests. The system manager can display the existing locks in the Lock Table or remove selected locks using the Management Portal interface or the ^LOCKTAB utility. You can also use the %SYS.LockQuery class to read lock table information. From the %SYS namespace you can use the SYS.Lock class to manage the lock table.

13.1.1 Viewing Locks Using the Lock Table

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**) 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 in the "Monitoring InterSystems IRIS Using the Management Portal" chapter of *Monitoring Guide*.

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:

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:

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:

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.

The **Routine** column provides the current line number and routine that the owner process is executing.

The **View Locks** window cannot be used to remove locks.

13.1.2 Removing Locks Using the Lock Table

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.

13.2 ^LOCKTAB Utility

You can also view and delete (remove) locks using the InterSystems IRIS **^LOCKTAB** utility from the %SYS namespace. 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.

13.3 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.

13.3.1 Queuing of Array Node Lock Requests

The InterSystems IRIS 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. As this may differ from expectations, or from implementations of lock queuing on other databases, some clarification is provided here.

Consider the case where three locks on the same global array are requested by three different processes:

```
Process A: LOCK ^x(1,1)
Process B: LOCK ^x(1)
Process C: LOCK ^x(1,2)
```

In this case, Process A gets a lock on ^x(1,1). Process B must wait for Process A to release ^x(1,1) before locking ^x(1). But what about Process C? The lock granted to Process A blocks Process B, but no held lock blocks the Process C lock request. It is the fact that Process B is waiting to explicitly lock ^x(1) and thus implicitly lock ^x(1,2) — which is the node that Process C wants to lock — that blocks Process C. In InterSystems IRIS, Process C must wait for Process B to lock and unlock.

The InterSystems IRIS lock queuing algorithm is fairest for Process B. Other database implementations that allowed Process C to jump the queue can speed Process C, but could (especially if there are many jobs such as Process C) result in an unacceptable delay for Process B.

Note: This strict process queuing algorithm applies to all subscripted lock requests. However, a process releasing a non-subscripted lock (such as LOCK -^abc) when there are both non-subscripted (LOCK +^abc) and subscripted (LOCK +^abc(1,1)) waiting lock requests is a special case. In this case, which lock request is serviced is unpredictable and may not follow strict process queuing.

13.3.2 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"	2. LOCK ^a(1)#"S"
3. LOCK +^a(1)	4. LOCK +^a(1)
Waits on release of Process B shared lock; deadlock.	Waits on release of Process A shared lock; deadlock.

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, you should either request the exclusive lock without the plus sign (thus unlocking 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"	2. LOCK ^a(1)#"S"
3. LOCK ^a(1)	4. LOCK ^a(1)
Releases shared lock, waits on Process B shared lock.	Releases shared lock, immediately applies exclusive 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, you should always include the *timeout* argument when using plus sign locks. For example, **LOCK +^a(1):10**.

If a deadlock occurs, you can resolve it by using the Management Portal or the **LOCKTAB** utility to remove one of the locks in question. From the Management Portal, open the **Locks** window, then select the Remove option for the deadlocked process.

14

Transaction Processing

This chapter covers the following topics:

- Managing Transactions Within Applications
  - LOCK in a Transaction
  - $INCREMENT and $SEQUENCE in a Transaction
- Automatic Transaction Rollback
- System-Wide Issues with Transaction Processing

A transaction is a logical unit of work. InterSystems IRIS® data platform transaction processing helps maintain the logical integrity of your database.

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.

Within your application, a single SQL INSERT, UPDATE, or DELETE statement, or a single global **SET** or **KILL**, may not in itself constitute a complete transaction. 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.

14.1 Managing Transactions Within Applications

In InterSystems IRIS, you define transactions within applications using either

- SQL statements in macro source routines

- ObjectScript commands

Both techniques work, regardless of whether the database modifications that constitute the transactions are performed with SQL **INSERT**, **UPDATE**, and **DELETE** statements or ObjectScript **SET** and **KILL** commands.

14.1.1 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 14–1: 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**.

14.1.2 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 the "Lock Management" chapter of this book.
- Lock operations that time out set $TEST. A value set in **$TEST** during a transaction does not roll back.

14.1.3 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.

14.1.4 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.

14.1.4.1 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:

```
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.

14.1.4.2 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.

14.1.5 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:

```
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"
}
```

14.2 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.

14.3 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 the chapter Backup and Restore in the *Data Integrity Guide*; for more information on issues related to ECP, see the appendix ECP Recovery Process, Guarantees, and Limitations in the "Horizontally Scaling for User Volume with Distributed Caching" chapter of the *Scalability Guide*.

14.3.1 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" chapter of the *Data Integrity Guide*
- Importance of Journals section of the "Backup and Recovery" chapter of the *Data Integrity Guide*

14.3.2 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.

14.4 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.

15

Error Processing

Managing the behavior of InterSystems IRIS® data platform when an error occurs is called *error processing* or *error handling*. Error processing performs one or more of 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 supports three types of error processing, which can be used simultaneously. These are:

- The TRY-CATCH mechanism
- %Status error code processing
- Traditional $ZTRAP error processing

Important: The preferred mechanism for ObjectScript error handling is the **TRY-CATCH** mechanism. The **$ZTRAP** mechanism is provided for a more traditional style of error handling.

15.1 The TRY-CATCH Mechanism

InterSystems IRIS supports a **TRY-CATCH** mechanism for handling errors. With this mechanism, 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*.

15.1.1 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.

15.1.2 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:

```
  #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:

```
#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
}
```

These system-supplied macros are further described in the "ObjectScript Macros and the Macro Preprocessor" chapter of this book.

15.1.3 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

15.1.4 Other Considerations with TRY-CATCH

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

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.

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.

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.

15.2 %Status Error Processing

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

- Successful method execution returns a %Status of 1.
- Failed method execution returns %Status as 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.

You can use %SYSTEM.Status class methods to inspect and manipulate %Status values.

InterSystems IRIS provides several options for displaying (writing) the %Status encoded string in different formats. For further details, refer to "Display (Write) Commands" in the "Commands" chapter of this manual.

In the following example, the %Prepare fails because of an error in the myquery text: "ZOP" should be "TOP". This error is detected by the **IsError()** method, and other %SYSTEM.Status methods display the error code and text:

```
  SET myquery = "SELECT ZOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF ($System.Status.IsError(status)) {
      WRITE "%Prepare failed",!
      DO StatusError() }
  ELSE {WRITE "%Prepare succeeded",!
        RETURN }
StatusError()
  WRITE "Error #",$System.Status.GetErrorCodes(status),!
  WRITE $System.Status.GetOneStatusText(status,1),!
  WRITE "end of error display"
  QUIT
```

The following example is the same as the previous, except that the status error is detected by the `$$$ISERR()` macro of the %occStatus include file. `$$$ISERR()` (and its inverse, `$$$ISOK()`) checks whether or not %Status=1. The error code is returned by the `$$$GETERRORCODE()` macro:

```
#Include %occStatus
  SET myquery = "SELECT ZOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET status = tStatement.%Prepare(myquery)
  IF $$$ISERR(status) {
      WRITE "%Prepare failed",!
      DO StatusError() }
  ELSE {WRITE "%Prepare succeeded",!
        RETURN}
StatusError()
  WRITE "Error #",$$$GETERRORCODE(status),!
  WRITE $System.Status.GetOneStatusText(status,1),!
  WRITE "end of error display"
  QUIT
```

These system-supplied macros are further described in the "ObjectScript Macros and the Macro Preprocessor" chapter of this book.

Some methods, such as **%New()**, generate, but do not return a %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** system variable, as shown in the following example.

```
  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.

15.2.1 Creating %Status Errors

You can invoke system-defined %Status errors from your own methods by using the **Error()** method. You specify the error number that corresponds to the error message you wish to return.

```
  WRITE "Here my method generates an error",!
  SET status = $System.Status.Error(20)
  WRITE $System.Status.GetErrorText(status),!
```

You can include %1, %2, and %3 parameters in the returned error message, as shown in the following example:

```
  WRITE "Here my method generates an error",!
  SET status = $System.Status.Error(214,"3","^fred","BedrockCode")
  WRITE $System.Status.GetErrorText(status),!
```

You can localize the error message to display in your preferred language.

```
  SET status = $System.Status.Error(30)
  WRITE "In English:",!
  WRITE $System.Status.GetOneStatusText(status,1,"en"),!
  WRITE "In French:",!
  WRITE $System.Status.GetOneStatusText(status,1,"fr"),!
```

For a list of error codes and messages (in English), refer to the "General Error Messages" chapter of the *InterSystems IRIS Error Reference*.

You can use the generic error codes 83 and 5001 to specify a custom message that does not correspond to any of the general error messages.

You can use the **AppendStatus()** method to create a list of multiple error messages. Then you can use **GetOneErrorText()** or **GetOneStatusText()** to retrieve individual error messages by their position in this list:

```
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),!
```

15.2.2 %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.

15.3 Traditional Error Processing

This section describes various aspects of traditional error processing with InterSystems IRIS. These include:

15.3.1 How Traditional Error Processing Works

For traditional error processing, InterSystems IRIS provides the functionality so that your application can 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.

15.3.1.1 Internal Error-Trapping Behavior

To get the full benefit of InterSystems IRIS error processing and the scoping issues surrounding the **$ZTRAP** special variable, 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:

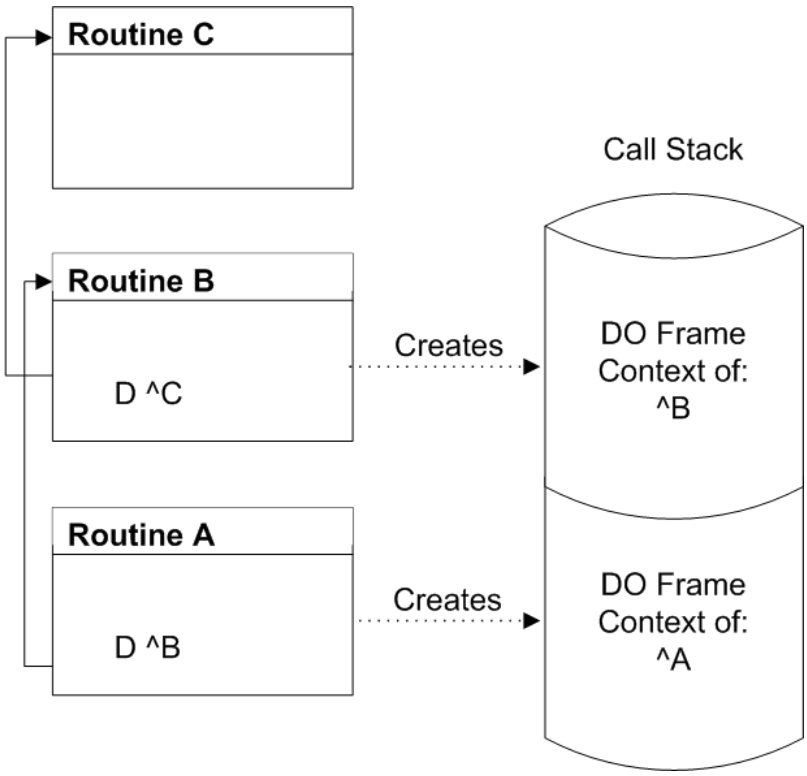
- A routine calls another routine with a **DO** command. (This kind of frame is also known as a "**DO** frame.")
- An **XECUTE** command argument causes ObjectScript code to execute. (This kind of frame is also known as a "**XECUTE** frame.")
- A user-defined function is executed.

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 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 15–1: 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.

15.3.1.2 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

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.

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.

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.

15.3.1.3 Error Codes

When an error occurs, InterSystems IRIS sets the **$ZERROR** and **$ECODE** special variables to a value describing the error.

$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:

```
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*.

$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.

15.3.2 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.

15.3.2.1 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*.

15.3.2.2 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:

```
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** entrypoint 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 entrypoints, see Logging Application Errors. For details on its use with **$ZTRAP**, see SET $ZTRAP=^%ETN.

For further details see the $ZTRAP special variable in the *ObjectScript Reference*.

15.3.2.3 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:

```
MyErrHandler
  SET $ZTRAP=""
  SET err=$ZERROR
  /* error handling code
     using err as the error
     to be handled */
```

15.3.2.4 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.

15.3.2.5 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:

```
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.

15.3.2.6 $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

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.

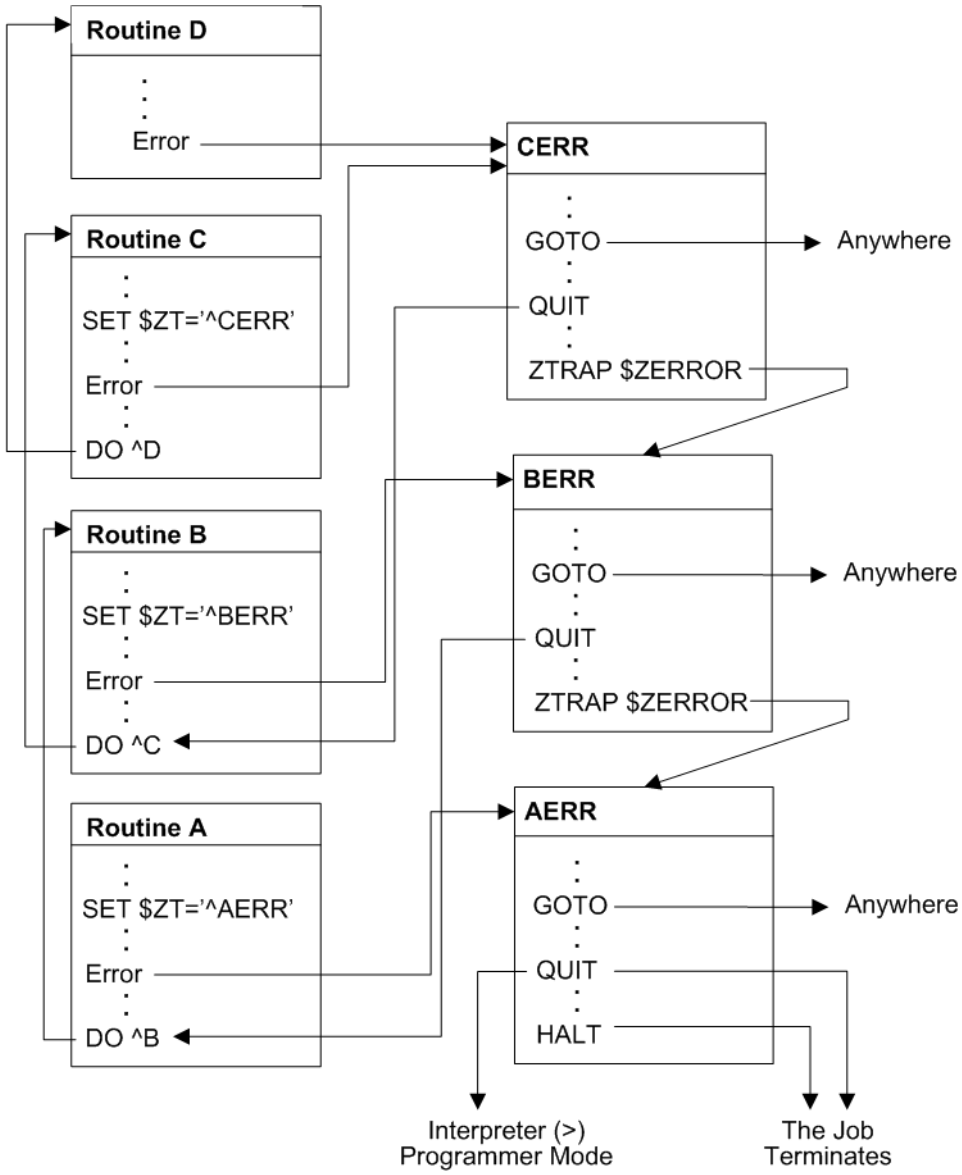
When an error handler has handled an error, the **$ZERROR** special variable is set to a value. This value is not necessarily cleared when the error handler completes. Some routines reset **$ZERROR** to the null string. The **$ZERROR** value is overwritten when the next error occurs that invokes an error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

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 $ZERROR` re-signals the error condition and causes InterSystems IRIS to unwind the call stack to the next call stack level with an error handler. After InterSystems IRIS has unwound the call stack to the level of the next error handler, processing continues in that error handler. The next error handler may have been set by a **$ZTRAP**.

The following figure shows the flow of control in **$ZTRAP** error handling routines.

Figure 15–2: $ZTRAP Error Handlers



15.3.3 Handling Errors in a $ZTRAP 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.

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.

15.3.3.1 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:

```
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*.

15.3.4 Forcing an Error

You set the **$ECODE** special variable or use the **ZTRAP** command to cause an error to occur under controlled circumstances.

15.3.4.1 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.

15.3.4.2 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".

```
SET $ECODE=",Upassword expired,"
```

15.3.5 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.

15.3.5.1 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:

```
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:

```
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.

15.3.5.2 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:

```
USER>

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:

```
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:

```
USER 4d0>
```

For a more detailed explanation, refer to Terminal Prompt Shows Program Stack Information in the "Command-line Routine Debugging" chapter.

15.3.5.3 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.

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:

```
USER>DO ^mytest
hello

  WRITE "hello",! SET x="world" SET y=zzz WRITE x,!
                                ^
<UNDEFINED>WriteOut+2^mytest *zzz
USER 2d0>GOTO
world

USER>
```

Resuming Execution at Another Line

You can resume execution at another line by issuing a **GOTO** with a label argument from the Terminal prompt:

```
USER 2d0>GOTO ErrSect
```

Deleting the Program Stack

You can delete the entire program stack by issuing an argumentless **QUIT** command from the Terminal prompt:

```
USER 4d0>QUIT
USER>
```

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:

```
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:

```
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.

15.4 Logging Application Errors

InterSystems IRIS provides several ways to log an exception to an application error log.

- The %ETN utility logs errors. It can be invoked as ^%ETN or using one of its entrypoints: FORE^%ETN, BACK^%ETN, or LOG^%ETN.
- The **%Exception.AbstractException.Log()** method.

15.4.1 Using %ETN to Log Application Errors

The **%ETN** utility logs an exception to the application error log and then exits. You can invoke **%ETN** (or one of its entrypoints) as a utility:

```
DO ^%ETN
```

Or you can set the $ZTRAP special variable to %ETN (or one of its entrypoints):

```
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:

```
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** entrypoints are invoked for a <STORE> error.

15.4.2 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. You can specify a user comment for an individual error.

The **Namespaces**, **Dates**, and **Errors** listings include checkboxes 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.

15.4.3 Using %ERN to View Application Error Logs

The %ERN utility examines application errors recorded by the %ETN error trap utility. %ERN returns all errors logged for the current namespace.

Take the following steps to use the %ERN utility:

1. At the Terminal prompt, **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. `For Date:` at this prompt, enter the date on which the errors occurred. You can use any date format that is accepted by the %DATE utility; if you omit the year, the current year is assumed. It returns 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.

3. `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.

4. `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 on the current Terminal device.
   - `*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 on the current Terminal device. %ERN returns the value of each specified variable (if defined) and all its descendent nodes.

16

Command-line Routine Debugging

This chapter describes techniques for testing and debugging InterSystems IRIS® data platform applications. Its topics include:

- Debugging with the ObjectScript Debugger
- Debugging With BREAK
- Using %STACK to Display the Stack
- Other Debugging Tools

An important part of application development is routine debugging: the testing and correcting of program code. InterSystems IRIS gives you two ways to debug your routines:

- 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.

16.1 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.

16.1.1 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. You can set a maximum of 20 watchpoints.

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.

16.1.2 Establishing Breakpoints and Watchpoints

You use the ZBREAK command to establish breakpoints and watchpoints.

16.1.2.1 Syntax

```
ZBREAK location[:action:condition:execute_code]
```

where:

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.
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* — Specifies an expression that will be evaluated when the breakpoint/watchpoint is triggered. The expression must be surrounded by quotation marks. If *condition* is false, the *action* will not be carried out and the *execute_code* will not be executed. If *condition* is not specified, the default is true.
execute_code	*Optional* — Specifies ObjectScript code to be executed if *condition* is true. The code must be surrounded by quotation marks if it is a literal. 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.

16.1.2.2 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.

16.1.2.3 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:

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:

label	Break before the line label at *label*.
label+3	Break before the third line after *label*.
+3	Break before the third line.

16.1.2.4 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:

$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.

16.1.2.5 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.

Argument	Description
"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.
"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.

16.1.2.6 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:

```
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.

```
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.

```
USER>DO ^test
FOR i=1:1:6 SET a=a+1
 ^
<BREAK>
test+3^test
USER 3f0>WRITE a
5
```

16.1.3 Disabling Breakpoints and Watchpoints

You can disable either:

- Specific breakpoints and watchpoints
- All breakpoints or watchpoints

16.1.3.1 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
```

16.1.3.2 Disabling All Breakpoints and Watchpoints

You can disable all breakpoints or watchpoints by using the plus or minus signs without a location:

ZBREAK -	Disable all defined breakpoints and watchpoints.
ZBREAK +	Enable all defined breakpoints and watchpoint.

16.1.4 Delaying Execution of Breakpoints and Watchpoints

You can also delay the execution of a break/watch point 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.

16.1.5 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.

16.1.6 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:

```
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:

```
USER>ZBREAK $:"TB"
```

16.1.7 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:

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.

16.1.7.1 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.

16.1.8 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**

16.1.9 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/watch point	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

16.1.9.1 Default Display

When you first enter InterSystems IRIS and use **ZB**, the output is as follows:

```
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

16.1.9.2 Display When Breakpoints and Watchpoints Exist

This example shows two breakpoints and one watchpoint being defined:

```
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.

16.1.10 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.

16.1.11 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:

```
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:

```
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.

16.1.12 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:

```
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:

```
USER>IF 1 WRITE b IF 1
```

16.2 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.)

16.2.1 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

   You can now issue ObjectScript commands, modify data, and execute further routines or subroutines, even those with errors or additional BREAKs.

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, refer to Command Postconditional Expressions in this manual.

16.2.2 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.

16.2.2.1 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:

```
ZBREAK /CLEAR
ZBREAK /TRACE:OFF
ZBREAK /DEBUG:""
ZBREAK /ERRORTRAP:ON
BREAK "C-"
```

16.2.3 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:

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 16–1: 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:

```
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:

```
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>
```

16.2.4 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.

16.2.5 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:

```
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:

```
USER 4f0>GOTO label3
```

You can also resume execution at the beginning of a line in a different routine:

```
USER 4f0>GOTO label3^rou
```

Alternatively, you may clear the program stack with an argumentless **QUIT** command:

```
USER 4f0>QUIT
USER>
```

16.2.5.1 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.

```
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.

```
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>
```

16.2.6 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:

```
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.

16.2.7 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:

```
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:

```
9f0>QUIT 3
6d0>
```

16.2.8 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.

16.3 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.

16.3.1 Running %STACK

You execute %STACK by entering the following command:

```
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.

16.3.2 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.

16.3.2.1 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.

16.3.2.2 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>
```

16.3.2.3 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.

16.3.2.4 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

16.3.3 Understanding the Stack Display

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

Table 16–2: %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. $$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 16–3: 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.			

16.3.3.1 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.

16.3.3.2 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:
```

16.3.3.3 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:
```

16.3.3.4 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()**
$ZU(23,1) = 5	
$ZU(34) = 0	
$ZU(39) = USER	**SysRoutinePath()**

Process State Intrinsics	Documentation
$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()**
$ZU(131,0) = MYCOMPUTER	
$ZU(131,1) = MYCOMPUTER:IRIS	

Process State Intrinsics	Documentation
$ZV = IRIS for Windows (x86-64) 2018.1.0 (Build 527U) Tue Feb 20 2018 22:47:10 EST	$ZVERSION special variable

16.3.3.5 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.

16.4 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

16.4.1 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 the section "OREF Basics" in the "Working with Registered Objects" chapter of *Defining and Using Classes*.

16.4.2 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.

16.4.2.1 %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")
```

16.4.2.2 %ERN Application Error Report

The %ERN utility examines application errors recorded by the %ETN error trap utility. For information on using %ERN, refer to Using %ERN to View Application Error Logs in the "Error Processing" chapter of this manual.

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.

```
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>
```