Using Multidimensional Storage (Globals)

Version 2023.3
2024-05-16

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

Using Multidimensional Storage (Globals)
InterSystems IRIS Data Platform Version 2023.3 2024-05-16

For Support questions about any InterSystems products, contact:

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

Table of Contents

1

Introduction to Globals

This topic introduces you to the concept of *globals*, the underlying multidimensional storage structure for InterSystems IRIS® data platform. No matter how you decide to store or access your data, what you're doing is using globals.

Globals can be accessed using a relational model, using an object model, or directly. For a video that explains the benefits of this multi-model access and a hands-on exercise that lets you try the three alternatives yourself, see Exploring Multiple Data Models with Globals.

1.1 What Are Globals?

One of the hallmarks of the InterSystems IRIS is its ability to store data once and allow you to access it using multiple paradigms. For example, you can use InterSystems SQL to visualize your data as rows and columns, or you can use ObjectScript and think of your data in terms of objects that have properties and methods. Your application can even mix these data models, using whichever model is easiest and more efficient for a given task. But no matter how you write or access your data, InterSystems IRIS stores it in underlying data structures known as globals.

Globals are persistent multidimensional sparse arrays:

- *Persistent* — Globals are stored in the database and can be retrieved at any time, by any process that can access that database.
- *Multidimensional* — The nodes in a global can have any number of subscripts. These subscripts can be integers, decimal numbers, or strings.
- *Sparse* — Node subscripts do not have to be contiguous, meaning that subscripts without a stored value do not use any storage.

Nodes in a global can store many types of data, including:

- Strings
- Numeric data
- Streams of character or binary data
- Collections of data, such as lists or arrays
- References to other storage locations

Even the server-side code you write is ultimately stored in globals!

1.2 Why Should Application Developers Learn About Globals?

While it is possible to write an application on the InterSystems IRIS platform with little or no knowledge of globals, there are several reasons why you may want to learn more about them:

- Some operations may be easier or more efficient if you access globals directly.
- You may want to create custom data structures for data that does not conform to relational or object data models.
- Some system administration tasks are done at the global level, and understanding globals will make these tasks more meaningful to you.

1.3 Examples of Globals

If you're new to InterSystems IRIS, you may be tempted to compare globals to data structures you have encountered from programming in other languages. This is a difficult exercise because a global is a flexible data structure that can be used in many different ways. But no matter the type of data it holds, a global is differentiated from a regular variable by placing a caret (^) in front of the name. This indicates that the variable is persisted to the database.

1.3.1 Scalars

In its simplest form, a global can be used to store a single value, or scalar:

```
^a = 4
```

In this example, the global ^a holds an integer with the value 4, but as mentioned earlier, it can hold data of other types just as easily.

1.3.2 Arrays

Globals can also be used as you would use an array in other languages, for example:

```
^month(1) = "January"
^month(2) = "February"
^month(3) = "March"
^month(4) = "April"
.
.
.
^month(12) = "December"
```

However, not every subscript in the array must include data. Since an array can be sparse, no storage is allocated for locations in an array that are not used.

```
^sparse(1,2,3) = 16
^sparse(1,2,5000) = 400
```

And, unlike arrays in many languages, the subscripts of a global can be negative numbers, real numbers, or strings. And the same array can hold data of varying types.

```
^misc(-4, "hello", 3.14) = 0
^misc("Sam", 27) = "Persimmon"
```

1.3.3 Dictionaries

Because of their flexibility, many people conceptualize globals as dictionaries (or nested dictionaries), with key-value pairs. In the following example, the global `^team` stores information about a baseball team:

```
^team("ballpark") = "Fenway Park"
^team("division") = "East"
^team("established") = 1901
^team("league") = "American"
^team("name") = "Boston Red Sox"
^team("retired number",1) = "Bobby Doerr"
^team("retired number",4) = "Joe Cronin"
^team("retired number",6) = "Johnny Pesky"
^team("retired number",8) = "Carl Yastrzemski"
^team("retired number",9) = "Ted Williams"
^team("world series titles") = $lb(1903,1912,1915,1916,1918,2004,2007,2013,2018)
```

In many languages, dictionaries are unordered, meaning that when you retrieve data from the dictionary, the data can be returned in some unspecified order. With globals, however, data is sorted according to its subscripts as it is stored.

1.3.4 Ordered Trees

It is more accurate to visualize a global as an ordered tree, where each node in the tree can have a value and/or children. In this regard, it is more flexible than nested dictionaries in other languages, where typically only the leaves of the tree contain data. In the following example, the global `^bird` stores birds according to their scientific names, with the names of each bird stored at the leaves of the tree. Here, the root node stores an overall description of the global, while a node representing a family of birds stores a description of that family:

```
^bird = "Birds of North America"
^bird("Anatidae") = "Ducks, Geese and Swans"
^bird("Anatidae", "Aix", "sponsa") = "Wood Duck"
^bird("Anatidae", "Anas", "rubripes") = "American Black Duck"
^bird("Anatidae", "Branta", "leucopsis") = "Barnacle Goose"
^bird("Odontophoridae") = "New World Quails"
^bird("Odontophoridae", "Callipepia", "californica") = "California Quail"
^bird("Odontophoridae", "Callipepia", "gambelii") = "Gambel's Quail"
```

For an animated illustration of how data is stored in ordered trees, see Ordered Trees.

For a short hands-on exercise and a whiteboard demonstration on globals, see Globals Quickstart.

1.4 Globals and External Languages

If you are writing an application in any of the supported external languages, InterSystems IRIS provides APIs that allow you to manipulate your data using the three models discussed in this topic, as follows:

- Relational access through JDBC, ADO.NET, DB-API, or ODBC
- Object access through the InterSystems XEP APIs for Java and .Net
- Direct access to globals through the InterSystems Native SDKs

Note: Not all forms of access are supported for all languages.

2

Formal Rules about Globals

This topic describes formal rules governing globals and global references (apart from extended global references, discussed separately).

2.1 Introduction to Global Names and Limits

The basic rules for global names are as follows:

- The name begins with a caret character (^) prefix. This caret distinguishes a global from a local variable.
- The next character can be a letter or the percent character (%):
  - Globals with names that start ^% are available in all namespaces. These are sometimes called percent globals.
  - Globals with names that do not use % are available only in the current namespace unless there are global mappings in effect.
- The other characters of a global name may be letters, numbers, or the period (.) character, except that the last character of the name cannot be a period.
- A global name may be up to 31 characters long (exclusive of the caret character prefix). You can specify global names that are significantly longer, but InterSystems IRIS treats only the first 31 characters as significant.
- Global names are case-sensitive.
- There are naming conventions to follow to avoid collision with InterSystems globals; see Global Variable Names to Avoid.
- InterSystems IRIS imposes a limit on the total length of a global reference, and this limit, in turn, imposes limits on the length of any subscript values. See Maximum Length of a Global Reference.

For more details, see Rules and Guidelines for Identifiers.

2.1.1 Variations

- A process-private global is an array variable that is only accessible to the process that created it. The name of a process-private global starts with ^|| rather than a single caret (^). For details, see Process-Private Globals.
- You can refer to a global in another namespace via an extended global reference.

2.2 Introduction to Global Nodes and Subscripts

A global typically has multiple *nodes*, generally identified by a *subscript* or set of subscripts. For a basic example:

ObjectScript

```
set ^Demo(1)="Cleopatra"
```

This statement refers to the global node `^Demo(1)`, which is a node within the `^Demo` global. This node is identified by one subscript.

For another example:

ObjectScript

```
set ^Demo("subscript1","subscript2","subscript3")=12
```

This statement refers to the global node `^Demo("subscript1","subscript2","subscript3")`, which is another node within the same global. This node is identified by three subscripts.

For yet another example:

ObjectScript

```
set ^Demo="hello world"
```

This statement refers to the global node `^Demo`, which does not use any subscripts.

The nodes of a global form a hierarchical structure. ObjectScript provides commands that take advantage of this structure. You can, for example, remove a node or remove a node and all its children; see Using Multidimensional Storage (Globals).

Important: Note that any global node cannot contain a string longer than the string length limit, which is extremely long. See General System Limits.

2.3 Rules for Global Subscripts

Subscripts have the following rules:

- Subscript values are case-sensitive.
- A subscript value can be any ObjectScript expression, provided that the expression does not evaluate to the null string ("").

  The value can include characters of all types, including blank spaces, non-printing characters, and Unicode characters. (Note that non-printing characters are less practical in subscript values.)

- Before resolving a global reference, InterSystems IRIS evaluates each subscript in the same way it evaluates any other expression. In the following example, we set one node of the ^Demo global, and then we refer to that node in several equivalent ways:

```
SAMPLES>s ^Demo(1+2+3)="a value"

SAMPLES>w ^Demo(3+3)
a value

SAMPLES>w ^Demo(03+03)
a value

SAMPLES>w ^Demo(03.0+03.0)
a value

SAMPLES>set x=6

SAMPLES>w ^Demo(x)
a value
```

- InterSystems IRIS imposes a limit on the total length of a global reference, and this limit, in turn, imposes limits on the length of any subscript values. See Maximum Length of a Global Reference.

CAUTION: The preceding rules apply for all InterSystems IRIS supported collations. For older collations still in use for compatibility reasons, such as "pre-ISM-6.1", the rules for subscripts are more restrictive. For example, character subscripts cannot have a control character as their initial character; and there are limitations on the number of digits that can be used in integer subscripts.

2.4 Collation of Globals

Within a global, nodes are stored in a collated (sorted) order.

Applications typically control the order in which nodes are sorted by applying a conversion to values used as subscripts. For example, the SQL engine, when creating an index on string values, converts all string values to uppercase letters and prepends a space character to make sure that the index is both not case-sensitive and collates as text (even if numeric values are stored as strings).

3

Extended Global References

You can refer to a global located in a namespace other than the current namespace. This is known as an *extended global reference* or simply an *extended reference*.

Note that the rule about the maximum length of a global reference applies to extended global references as well as to the more common global references.

3.1 Forms of Extended Global References

There are two forms of extended references:

- Explicit namespace reference — You specify the name of the namespace where the global is located as part of the syntax of the global reference.
- Implied namespace reference — You specify the directory and, optionally, the system name as part of the syntax of the global reference. In this case, no global mappings apply, since the physical dataset (directory and system) is given as part of the global reference.

The use of explicit namespaces is preferred, because this allows for redefinition of logical mappings externally, as requirements change, without altering your application code.

InterSystems IRIS supports two syntaxes for extended references:

- Bracket syntax, which encloses the extended reference with square brackets ([ ]).
- Environment syntax, which encloses the extended reference with vertical bars (| |).

Note: The examples shown here use the Windows directory structure. In practice, the form of such references is operating-system dependent.

3.2 Bracket Syntax

You can use bracket syntax to specify an extended global reference with either an explicit namespace or an implied namespace:

Explicit namespace:

```
^[nspace]glob
```

Implied namespace:

```
^[dir,sys]glob
```

In an explicit namespace reference, *nspace* is a defined namespace that the global *glob* has not currently been mapped or replicated to. In an implied namespace reference, *dir* is a directory (the name of which includes a trailing backslash: \), *sys* is a system, and *glob* is a global within that directory. If *nspace* or *dir* is specified as a carat (^), the reference is to a process-private global.

You must include quotation marks around the directory and system names or the namespace name unless you specify them as variables. The directory and system together comprise an implied namespace. An implied namespace can reference either:

- The specified directory on the specified system.
- The specified directory on your local system, if you do not specify a system name in the reference. If you omit the system name from an implied namespace reference, you must supply a double caret (^^) within the directory reference to indicate the omitted system name.

To specify an implied namespace on a remote system:

```
["dir","sys"]
```

To specify an implied namespace on the local system:

```
["^^dir"]
```

For example, to access the global SAMPLE in the C:\BUSINESS\ directory on a machine called SALES:

ObjectScript

```
Set x = ^["C:\BUSINESS\","SALES"]SAMPLE
```

To access the global SAMPLE in the C:\BUSINESS\ directory on your local machine:

ObjectScript

```
Set x = ^["^^C:\BUSINESS\"]SAMPLE
```

To access the global SAMPLE in the defined namespace MARKETING:

ObjectScript

```
Set x = ^["MARKETING"]SAMPLE
```

To access the process-private global SAMPLE:

ObjectScript

```
Set x = ^["^"]SAMPLE
```

3.3 Bracket Syntax with References to Databases

InterSystems IRIS provides special bracket syntaxes to represent databases within extended references.

You can create an extended reference that includes a database name, as specified in the CPF file. Use the format `:ds:`*DB_name*. For example

```
["^^:ds:MYDATABASE"]
```

A similar syntax is available for an extended reference that refers to a database on a mirror. Use the format `:mirror:`*mirror_name*:*mirror_DB_name*. For example, when referring to the database with the mirror database name mirdb1 in the mirror CORPMIR, you could form an implied reference as follows:

```
["^^:mirror:CORPMIR:mirdb1"]
```

The mirrored database path can be used for both local and remote databases.

3.4 Environment Syntax

The environment syntax is defined as:

```
^|"env"|global
```

"env" can have one of five formats:

- The null string (`""`) — The current namespace on the local system.
- `"namespace"` — A defined namespace that *global* is not currently mapped to. Namespace names are not case-sensitive. If *namespace* has the special value of `"^"`, it is a process-private global.
- `"^^dir"` — An implied namespace whose default directory is the specified directory on your local system, where *dir* includes a trailing backslash (\).
- `"^system^dir"` — An implied namespace whose default directory is the specified directory on the specified remote system, where *dir* includes a trailing backslash (\).
- omitted — If there is no *"env"* at all, it is a process-private global.

To access the global SAMPLE in your current namespace on your current system, when no mapping has been defined for SAMPLE, use the following syntax:

ObjectScript

```
Set x = ^|""|SAMPLE
```

This is the same as the simple global reference:

ObjectScript

```
Set x = ^SAMPLE
```

To access the global SAMPLE mapped to the defined namespace MARKETING:

ObjectScript

```
Set x = ^|"MARKETING"|SAMPLE
```

You can use an implied namespace to access the global SAMPLE in the directory C:\BUSINESS\ on your local system:

ObjectScript

```
Set x = ^|"^^C:\BUSINESS\"|SAMPLE
```

You can use an implied namespace to access the global SAMPLE in the directory C:\BUSINESS on a remote system named SALES:

ObjectScript

```
Set x = ^|"^SALES^C:\BUSINESS\"|SAMPLE
```

To access the process-private global SAMPLE:

ObjectScript

```
Set x = ^||SAMPLE
Set x=^|"^"|SAMPLE
```

4

Global Mapping and Subscript-Level Mapping

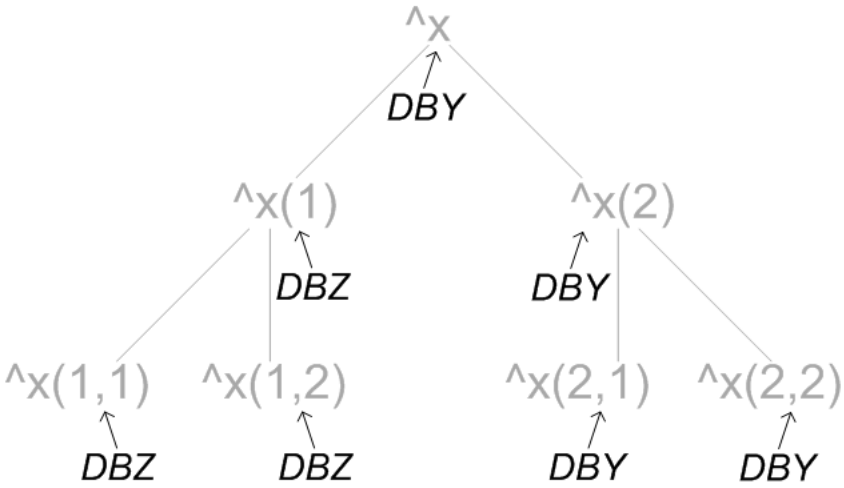
You can map globals and routines from one database to another on the same or different systems. This allows simple references to data which can exist anywhere and is the primary feature of a namespace. You can map whole globals or pieces of globals; mapping a piece of a global (or a subscript) is known as *subscript-level mapping (SLM)*.

You can map globals and routines from one database to another on the same or different systems. Because you can map global subscripts, data can easily span disks.

To configure this type of mapping, see Add Global, Routine, and Package Mapping to a Namespace.

4.1 Simple Example of Subscript-Level Mapping

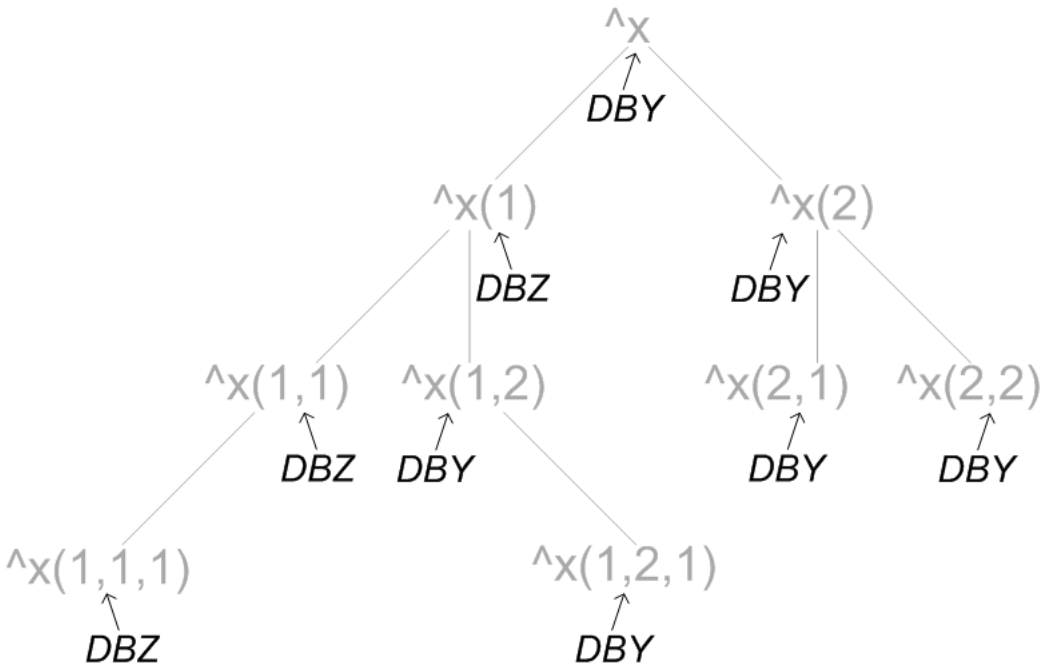
Global mapping is applied hierarchically. For example, if the NSX namespace has an associated DBX database, but maps the ^*x* global to the DBY database and ^*x(1)* to the DBZ database, then any subscripted form of the ^x global — except those that are part of the ^*x(1)* hierarchy — is mapped to DBY; those globals that are part of the ^*x(1)* hierarchy are mapped to DBZ. The following diagram illustrates this hierarchy:



In this diagram, the globals and their hierarchy appear in gray, and the databases to which they are mapped appear in black.

4.2 More Complex Example of Subscript-Level Mapping

It is also possible to map part of a mapped, subscripted global to another database, or even back to the database to which the initial global is mapped. Suppose that the previous example had the additional mapping of the *^x(1,2)* global back to the DBY database. This would appear as follows:



Again, the globals and their hierarchy appear in gray, and the databases to which they are mapped appear in black.

Once you have mapped a global from one namespace to another, you can reference the mapped global as if it were in the current namespace — with a simple reference, such as *^ORDER* or *^X(1)*.

Important: When establishing subscript-level mapping ranges, the behavior of string subscripts differs from that of integer subscripts. For strings, the first character determines the range, while the range for integers uses numeric values. For example, a subscript range of ("A"):("C") contains not only AA but also AC and ABCDEF; by contrast, a subscript range of (1):(2) does not contain 11.

4.3 Key Principles

4.3.1 Using Distinct Ranges of Globals and Subscripts

Each of a namespace's mappings must refer to distinct ranges of globals or subscripts. Mapping validation prevents the establishment of any kind of overlap. For example, if you attempt to use the Management Portal to create a new mapping that overlaps with an existing mapping, the Portal prevents this from occurring and displays an error message.

4.3.2 Logging Changes

Successful changes to the mappings through the Portal are also logged in messages.log; unsuccessful changes are not logged. Any failed attempts to establish mappings by hand-editing the configuration parameter (CPF) file are logged in messages.log; for details on editing the CPF, see Editing the Active CPF.

5

Working with Globals

This topic describes the various operations you can perform using multidimensional storage (globals).

Also see Temporary Globals and the IRISTEMP Database.

Note: When using direct global access within applications, develop and adhere to a naming convention to keep different parts of an application from "walking over" one another; this is similar to developing naming convention for classes, method, and other variables. Also, avoid certain global names that InterSystems IRIS® data platform uses; for a list of these, see Global Variable Names to Avoid.

5.1 Storing Data in Globals

Storing data in global nodes is simple: you treat a global as you would any other variable. The difference is that operations on globals are automatically written to the database.

5.1.1 Creating Globals

There is no setup work required to create a new global; simply setting data into a global implicitly creates a new global structure. You can create a global (or a global subscript) and place data in it with a single operation, or you can create a global (or subscript) and leave it empty by setting it to the null string. In ObjectScript, these operations are done using the **SET** command.

The following examples define a global named *Color* (if one does not already exist) and associate the value "Red" with it. If a global already exists with the name *Color*, then these examples modify it to contain the new information.

In ObjectScript:

ObjectScript

```
SET ^Color = "Red"
```

5.1.2 Storing Data in Global Nodes

To store a value within a global subscript node, simply set the value of the global node as you would any other variable. If the specified node did not previously exist, it is created. If it did exist, its contents are replaced with the new value.

You can store any data in a global node, with the exception that any global node cannot contain a string longer than the string length limit, which is extremely long. See General System Limits.

Setting the value of a global node is an *atomic* operation: It is guaranteed to succeed and you do not need to use any locks to ensure concurrency.

ObjectScript

```
SET ^TEST = 2
SET ^TEST("Color")="Red"
SET ^TEST(1,1)=100        /* The 2nd-level subscript (1,1) is set
                             to the value 100. No value is stored at
                             the 1st-level subscript (^TEST(1)). */
SET ^TEST(^TEST)=10       /* The value of global variable ^TEST
                             is the name of the subscript. */
SET ^TEST(a,b)=50         /* The values of local variables a and b
                             are the names of the subscripts. */
SET ^TEST(a+10)=50
```

Also, you can construct global references at runtime using indirection.

5.2 Deleting Global Nodes

To remove a global node, a group of subnodes, or an entire global from the database, use the ObjectScript **KILL** or **ZKILL** commands.

The **KILL** command deletes all nodes (data as well as its corresponding entry in the array) at a specific global reference, including any descendant subnodes. That is, all nodes starting with the specified subscript are deleted.

For example, the ObjectScript statement:

ObjectScript

```
KILL ^TEST
```

deletes the entire ^TEST global. A subsequent reference to this global would return an <UNDEFINED> error.

The ObjectScript statement:

ObjectScript

```
KILL ^TEST(100)
```

deletes contents of node 100 within the ^TEST global. If there are descendant subnodes, such as ^TEST(100,1), ^TEST(100,2), and ^TEST(100,1,2,3), these are deleted as well.

The ObjectScript **ZKILL** command deletes a specified global or global subscript node. It does not delete descendant subnodes.

Note: Following the kill of a large global, the space once occupied by that global may not have been completely freed, since the blocks are marked free in the background by the Garbage Collector daemon. Thus, a call to the **ReturnUnusedSpace** method of the SYS.Database class immediately after killing a large global may not return as much space as expected, since blocks occupied by that global may not have been released as yet.

You cannot use the **NEW** command on global variables.

5.3 Testing the Existence of a Global Node

To test if a specific global (or its descendants) contains data, use the **$DATA** function.

$DATA returns a value indicating whether or not the specified global reference exists. The possible return values are:

Status Value	Meaning
0	The global variable is undefined.
1	The global variable exists and contains data, but has no descendants. Note that the null string ("") qualifies as data.
10	The global variable has descendants (contains a downward pointer to a subnode) but does not itself contain data. Any direct reference to such a variable will result in an <UNDEFINED> error. For example, if $DATA(^y) returns 10, SET x=^y will produce an <UNDEFINED> error.
11	The global variable both contains data and has descendants (contains a downward pointer to a subnode).

5.4 Retrieving the Value of a Global Node

To get the value stored within a specific global node, simply use the global reference as an expression:

ObjectScript

```
SET color = ^TEST("Color")    ; assign to a local variable
WRITE ^TEST("Color")          ; use as a command argument
SET x=$LENGTH(^TEST("Color")) ; use as a function parameter
```

5.4.1 The $GET Function

You can also get the value of a global node using the **$GET** function:

ObjectScript

```
SET mydata = $GET(^TEST("Color"))
```

This retrieves the value of the specified node (if it exists) or returns the null string ("") if the node has no value. You can use the optional second argument of **$GET** to return a specified default value if the node has no value.

5.4.2 The WRITE, ZWRITE, and ZZDUMP Commands

You can display the contents of a global or a global subnode by using the various ObjectScript display commands. The **WRITE** command returns the value of the specified global or subnode as a string. The **ZWRITE** command returns the name of the global variable and its value, and each of its descendant nodes and their values. The **ZZDUMP** command returns the value of the specified global or subnode in hexadecimal dump format.

5.5 Traversing Data within a Global

There are a number of ways to traverse (iterate over) data stored within a global.

5.5.1 The $ORDER (Next / Previous) Function

The ObjectScript **$ORDER** function allows you to sequentially visit each node within a global.

Given a subscript (or set of subscripts) as an argument, the **$ORDER** function returns the value of the next subscript at a given level. This is best explained by example. Suppose you have defined a set of nodes in a global named ^TEST, as follows:

ObjectScript

```
Set ^TEST(1) = ""
Set ^TEST(1,1) = ""
Set ^TEST(1,2) = ""
Set ^TEST(2) = ""
Set ^TEST(2,1) = ""
Set ^TEST(2,2) = ""
Set ^TEST(5,1,2) = ""
```

To find the first, first-level subscript, we can use:

ObjectScript

```
SET key = $ORDER(^TEST(""))
```

This returns the first, top-level subscript following the null string (""). (The null string is used to represent the subscript value *before* the first entry; as a return value it is used to indicate that there are no following subscript values.) In this example, *key* will now contain the value 1.

We can find the next, top-level subscript by using 1 or *key* in the **$ORDER** expression:

ObjectScript

```
SET key = $ORDER(^TEST(key))
```

If *key* has an initial value of 1, then this statement will set it to 2 (because ^TEST(2) is the next first-level subscript). Executing this statement again will set *key* to 5 as that is the next first-level subscript. Note that 5 is returned even though there is no data stored directly at ^TEST(5). Executing this statement one more time will set *key* to the null string (""), indicating that there are no more first level subscripts.

By using additional subscripts with the **$ORDER** function, you can iterate over different subscript levels. For example, using the data above, the statement:

ObjectScript

```
SET key = $ORDER(^TEST(1,""))
```

will set *key* to 1 because ^TEST(1,1) is the next second-level subscript. Executing this statement again will set *key* to 2 as that is the next second-level subscript. Executing this statement one more time will set *key* to "" indicating that there are no more second-level subscripts under node ^TEST(1).

5.5.1.1 Looping with $ORDER

The following ObjectScript code defines a simple global and then loops over all of its first-level subscripts:

ObjectScript

```
// clear ^TEST in case it has data
Kill ^TEST

// fill in ^TEST with sample data
For i = 1:1:100 {
    // Set each node to a random person's name
    Set ^TEST(i) = ##class(%PopulateUtils).Name()
```

```
}

// loop over every node
// Find first node
Set key = $Order(^TEST(""))

While (key '= "") {
    // Write out contents
    Write "#", key, " ", ^TEST(key),!

    // Find next node
    Set key = $Order(^TEST(key))
}
```

5.5.1.2 Additional $ORDER Arguments

The ObjectScript **$ORDER** function takes optional second and third arguments. The second argument is a direction flag indicating in which direction you wish to traverse a global. The default, 1, specifies forward traversal, while –1 specifies backward traversal.

The third argument, if present, contains a local variable name. If the node found by **$ORDER** contains data, the data found is written into this local variable. When you are looping over a global and you are interested in node values as well as subscript values, this approach is efficient and requires the fewest coding steps.

5.5.2 Looping Over a Global

If you know that a given global is organized using contiguous numeric subscripts, you can use a simple For loop to iterate over its values. For example:

ObjectScript

```
For i = 1:1:100 {
    Write ^TEST(i),!
}
```

Generally, it is better to use the **$ORDER** function described above: it is more efficient and you do not have to worry about gaps in the data (such as a deleted node).

5.5.3 The $QUERY Function

If you need to visit every node and subnode within a global, moving up and down over subnodes, use the ObjectScript **$QUERY** function. (Alternatively you can use nested **$ORDER** loops).

The **$QUERY** function takes a global reference and returns a string containing the global reference of the next node in the global (or "" if there are no following nodes). To use the value returned by **$QUERY**, you must use the ObjectScript indirection operator (@).

For example, suppose you define the following global:

ObjectScript

```
Set ^TEST(1) = ""
Set ^TEST(1,1) = ""
Set ^TEST(1,2) = ""
Set ^TEST(2) = ""
Set ^TEST(2,1) = ""
Set ^TEST(2,2) = ""
Set ^TEST(5,1,2) = ""
```

The following call to **$QUERY**:

ObjectScript

```
SET node = $QUERY(^TEST(""))
```

sets *node* to the string "^TEST(1)", the address of the first node within the global. Then, to get the next node in the global, call **$QUERY** again and use the indirection operator on *node*:

ObjectScript

```
SET node = $QUERY(@node)
```

At this point, *node* contains the string "^TEST(1,1)".

The following example defines a set of global nodes and then walks over them using **$QUERY**, writing the address of each node as it does:

ObjectScript

```
Kill ^TEST // make sure ^TEST is empty

// place some data into ^TEST
Set ^TEST(1) = ""
Set ^TEST(1,1) = ""
Set ^TEST(1,2) = ""
Set ^TEST(2) = ""
Set ^TEST(2,1) = ""
Set ^TEST(2,2) = ""
Set ^TEST(5,1,2) = ""

// now walk over ^TEST
// find first node
Set node = $Query(^TEST(""))
While (node '= "") {
    Write node,!
    // get next node
    Set node = $Query(@node)
}
```

5.6 Copying Data within Globals

To copy the contents of a global (entire or partial) into another global (or a local array), use the ObjectScript **MERGE** command.

The following example demonstrates the use of the **MERGE** command to copy the entire contents of the `^OldData` global into the `^NewData` global:

ObjectScript

```
Merge ^NewData = ^OldData
```

If the source argument of the **MERGE** command has subscripts then all data in that node and its descendants are copied. If the destination argument has subscripts, then the data is copied using the destination address as the top level node. For example, the following code:

ObjectScript

```
Merge ^NewData(1,2) = ^OldData(5,6,7)
```

copies all the data at and beneath `^OldData(5,6,7)` into `^NewData(1,2)`.

5.7 Maintaining Shared Counters within Globals

A major concurrency bottleneck of large-scale transaction processing applications can be the creation of unique identifier values. For example, consider an order processing application in which each new invoice must be given a unique identifying number. The traditional approach is to maintain some sort of counter table. Every process creating a new invoice waits to acquire a lock on this counter, increments its value, and unlocks it. This can lead to heavy resource contention over this single record.

To deal with this issue, InterSystems IRIS provides the ObjectScript **$INCREMENT** function. **$INCREMENT** atomically increments the value of a global node (if the node has no value, it is set to 1). The atomic nature of **$INCREMENT** means that no locks are required; the function is guaranteed to return a new incremented value with no interference from any other process.

You can use **$INCREMENT** as follows. First, you must decide upon a global node in which to hold the counter. Next, whenever you need a new counter value, simply invoke **$INCREMENT**:

ObjectScript

```
SET counter = $INCREMENT(^MyCounter)
```

For persistent classes (other than those created via SQL, the default storage structure uses **$INCREMENT** to assign unique object (row) identifier values. For persistent classes created via SQL, the default storage structure instead uses $SEQUENCE.

5.8 Sorting Data within Globals

Data stored within globals is automatically sorted according to the value of the subscripts. For example, the following ObjectScript code defines a set of globals (in random order) and then iterates over them to demonstrate that the global nodes are automatically sorted by subscript:

ObjectScript

```
// Erase any existing data
Kill ^TEST

// Define a set of global nodes
Set ^TEST("Cambridge") = ""
Set ^TEST("New York") = ""
Set ^TEST("Boston") = ""
Set ^TEST("London") = ""
Set ^TEST("Athens") = ""

// Now iterate and display (in order)
Set key = $Order(^TEST(""))
While (key '= "") {
    Write key,!
    Set key = $Order(^TEST(key)) // next subscript
}
```

Applications can take advantage of the automatic sorting provided by globals to perform sort operations or to maintain ordered, cross-referenced indexes on certain values. InterSystems SQL and ObjectScript use globals to perform such tasks automatically.

5.8.1 Collation of Global Nodes

The order in which the nodes of a global are sorted (referred to as collation) is controlled at two levels: within the global itself and by the application using the global.

At the application level, you can control how global nodes are collated by performing data transformations on the values used as subscripts (InterSystems SQL and objects do this via user-specified collation functions). For example, if you wish to create a list of names that is sorted alphabetically but ignores case, then typically you use the uppercase version of the name as a subscript:

ObjectScript

```
// Erase any existing data
Kill ^TEST

// Define a set of global nodes for sorting
For name = "Cobra","jackal","zebra","AARDVark" {
    // use UPPERCASE name as subscript
    Set ^TEST($ZCONVERT(name,"U")) = name
}

// Now iterate and display (in order)
Set key = $Order(^TEST(""))
While (key '= "") {
    Write ^TEST(key),!  // write untransformed name
    Set key = $Order(^TEST(key)) // next subscript
}
```

This example converts each name to uppercase (using the $ZCONVERT function) so that the subscripts are sorted without regard to case. Each node contains the untransformed value so that the original value can be displayed.

5.8.2 Numeric and String-Valued Subscripts

Numeric values are collated *before* string values; that is a value of 1 comes before a value of “a”. You need to be aware of this fact if you use both numeric and string values for a given subscript. If you are using a global for an index (that is, to sort data based on values), it is most common to either sort values as numbers (such as salaries) or strings (such as postal codes).

For numerically collated nodes, the typical solution is to coerce subscript values to numeric values using the unary + operator. For example, if you are building an index that sort *id* values by *age*, you can coerce *age* to always be numeric:

ObjectScript

```
Set ^TEST(+age,id) = ""
```

If you wish to sort values as strings (such as “0022”, “0342”, “1584”) then you can coerce the subscript values to always be strings by prepending a space (“ ”) character. For example, if you are building an index that sort *id* values by *zipcode*, you can coerce *zipcode* to always be a string:

ObjectScript

```
Set ^TEST(" "_zipcode,id) = ""
```

This ensures that values with leading zeroes, such as “0022” are always treated as strings.

5.8.3 The $SORTBEGIN and $SORTEND Functions

Typically you do not have to worry about sorting data within InterSystems IRIS. Whether you use SQL or direct global access, sorting is handled automatically.

There are, however, certain cases where sorting can be done more efficiently. Specifically, in cases where (1) you need to set a large number of global nodes that are in random (that is, unsorted) order and (2) the total size of the resulting global approaches a significant portion of the InterSystems IRIS buffer pool, then performance can be adversely affected — since many of the **SET** operations involve disk operations (as data does not fit in the cache). This scenario usually arises in cases involving the creation of index globals such as bulk data loads, index population, or sorting of unindexed values in temporary globals.

To handle these cases efficiently, ObjectScript provides the **$SORTBEGIN** and **$SORTEND** functions. The **$SORTBEGIN** function initiates a special mode for a global (or part thereof) in which data set into the global is written to a special scratch buffer and sorted in memory (or temporary disk storage). When the **$SORTEND** function is called at the end of the operation, the data is written to actual global storage sequentially. The overall operation is much more efficient as the actual writing is done in an order requiring far fewer disk operations.

The **$SORTBEGIN** function is quite easy to use; simply invoke it with the name of the global you wish to sort before beginning the sort operation and call **$SORTEND** when the operation is complete:

ObjectScript

```
// Erase any existing data
Kill ^TEST

// Initiate sort mode for ^TEST global
Set ret = $SortBegin(^TEST)

// Write random data into ^TEST
For i = 1:1:10000 {
    Set ^TEST($Random(1000000)) = ""
}

Set ret = $SortEnd(^TEST)

// ^TEST is now set and sorted

// Now iterate and display (in order)
Set key = $Order(^TEST(""))
While (key '= "") {
    Write key,!
    Set key = $Order(^TEST(key)) // next subscript
}
```

The **$SORTBEGIN** function is designed for the special case of global creation and must be used with some care. Specifically, you must not *read* from the global to which you are writing while in **$SORTBEGIN** mode; as the data is not written, reads will be incorrect.

InterSystems SQL automatically uses these functions for creation of temporary index globals (such as for sorting on unindexed fields).

5.9 Using Indirection with Globals

By means of indirection, ObjectScript provides a way to create global references at runtime. This can be useful in applications where you do not know global structure or names at program compilation time.

Indirection is supported via the indirection operator, **@**, which de-references a string containing an expression. There are several types of indirection, based on how the **@** operator is used.

The following code provides an example of name indirection in which the **@** operator is used to de-reference a string containing a global reference:

ObjectScript

```
// Erase any existing data
Kill ^TEST

// Set var to an global reference expression
Set var = "^TEST(100)"

// Now use indirection to set ^TEST(100)
Set @var = "This data was set indirectly."

// Now display the value directly:
Write "Value: ",^TEST(100)
```

You can also use subscript indirection to mix expressions (variables or literal values) within indirect statements:

ObjectScript

```
// Erase any existing data
Kill ^TEST

// Set var to a subscript value
Set glvn = "^TEST"

// Now use indirection to set ^TEST(1) to ^TEST(10)
For i = 1:1:10 {
    Set @glvn@(i) = "This data was set indirectly."
}

// Now display the values directly:
Set key = $Order(^TEST(""))
While (key '= "") {
    Write "Value ",key, ": ", ^TEST(key),!
    Set key = $Order(^TEST(key))
}
```

Indirection is a fundamental feature of ObjectScript; it is not limited to global references. For more information, see Indirection. Indirection is less efficient than direct access, so you should use it judiciously.

5.10 Managing Concurrency

The operation of setting or retrieving a single global node is atomic; it is guaranteed to always succeed with consistent results. For operations on multiple nodes, InterSystems IRIS provides the ability to acquire and release locks. See Locking and Concurrency Control.

5.11 Checking the Most Recent Global Reference

The most recent global reference is recorded in the ObjectScript **$ZREFERENCE** special variable. **$ZREFERENCE** contains the most recent global reference, including subscripts and extended global reference, if specified. Note that **$ZREFERENCE** indicates neither whether the global reference succeeded, nor if the specified global exists. InterSystems IRIS simply records the most recently specified global reference.

5.11.1 Naked Global Reference

Following a subscripted global reference, InterSystems IRIS sets a *naked indicator* to that global name and subscript level. You can then make subsequent references to the same global and subscript level using a *naked global reference*, omitting the global name and higher level subscripts. This streamlines repeated references to the same global at the same (or lower) subscript level.

Specifying a lower subscript level in a naked reference resets the naked indicator to that subscript level. Therefore, when using naked global references, you are always working at the subscript level established by the most recent global reference.

The naked indicator value is recorded in the **$ZREFERENCE** special variable. The naked indicator is initialized to the null string. Attempting a naked global reference when the naked indicator is not set results in a <NAKED> error. Changing namespaces reinitializes the naked indicator. You can reinitialize the naked indicator by setting **$ZREFERENCE** to the null string ("").

In the following example, the subscripted global ^Produce(“fruit”,1) is specified in the first reference. InterSystems IRIS saves this global name and subscript in the naked indicator, so that the subsequent naked global references can omit the

global name “Produce” and the higher subscript level “fruit”. When the ^(3,1) naked reference goes to a lower subscript level, this new subscript level becomes the assumption for any subsequent naked global references.

ObjectScript

```
SET ^Produce("fruit",1)="Apples"  /* Full global reference  */
SET ^(2)="Oranges"                /* Naked global references */
SET ^(3)="Pears"                  /* assume subscript level 2 */
SET ^(3,1)="Bartlett pears"       /* Go to subscript level 3  */
SET ^(2)="Anjou pears"            /* Assume subscript level 3 */
WRITE "latest global reference is: ",$ZREFERENCE,!
ZWRITE ^Produce
KILL ^Produce
```

This example sets the following global variables: ^Produce("fruit",1), ^Produce("fruit",2), ^Produce("fruit",3), ^Produce("fruit",3,1), and ^Produce("fruit",3,2).

With few exceptions, every global reference (full or naked) sets the naked indicator. The **$ZREFERENCE** special variable contains the full global name and subscripts of the most recent global reference, even if this was a naked global reference. The **ZWRITE** command also displays the full global name and subscripts of each global, whether or not it was set using a naked reference.

Naked global references should be used with caution, because InterSystems IRIS sets the naked indicator in situations that are not always obvious, including the following:

- A full global reference initially sets the naked indicator, and subsequent full global references or naked global references change the naked indicator, even when the global reference is not successful. For example attempting to **WRITE** the value of a nonexistent global sets the naked indicator.
- A command postconditional that references a subscripted global sets the naked indicator, regardless of how InterSystems IRIS evaluates the postconditional.
- An optional function argument that references a subscripted global may or may not set the naked indicator, depending on whether InterSystems IRIS evaluates all arguments. For example the second argument of **$GET** always sets the naked indicator, even when the default value it contains is not used. InterSystems IRIS evaluates arguments in left-to-right sequence, so the last argument may reset the naked indicator set by the first argument.
- The **TROLLBACK** command, which rolls back a transaction, does not roll back the naked indicator to its value at the beginning of the transaction.

If a full global reference contains an extended global reference, subsequent naked global references assume the same extended global reference; you do not have to specify the extended reference as part of a naked global reference.

6

SQL and Persistent Class Use of Multidimensional Storage

This topic describes how InterSystems IRIS® data platform persistent classes and SQL engine make use of multidimensional storage (globals) for storing persistent objects, relational tables, and indexes.

Though the InterSystems IRIS object and SQL engines automatically provide and manage data storage structures, it can be useful to understand the details of how this works.

The storage structures used by the object and relational view of data are identical. For simplicity, this document only describes storage from the object perspective.

6.1 Storage Definitions

Every persistent class that uses the %Storage.Persistent storage class (the default) can store instances of itself within the InterSystems IRIS database using one or more nodes of multidimensional storage (globals). Specifically, every persistent class has a *storage definition* that defines how its properties are stored within global nodes. This storage definition (referred to as "default structure") is managed automatically by the class compiler. (You can modify this storage definition or even provide alternate versions of it if you like. This is not discussed in this document.)

6.1.1 Default Structure

The default structure used for storing persistent objects is quite simple:

- Data is stored in a global whose name starts with the complete class name, including package name. A `D` is appended to form the name of the data global, while an `I` is appended for the index global.

  (See Hashed Global Names for an option that results in a shorter global name.)

- Data for each instance is stored within a single node of the data global with all non-transient properties placed within a **$List** structure.

- Each node in the data global is subscripted by object ID value. For persistent classes (other than those created via SQL), the default storage structure uses **$Increment** to assign unique object (row) identifier values. For persistent classes created via SQL, the default storage structure instead uses **$Sequence**.

For example, suppose we define a simple persistent class, MyApp.Person, with two literal properties:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
}
```

If we create and save two instances of this class, the resulting global will be similar to:

```
^MyApp.PersonD = 2  // counter node
^MyApp.PersonD(1) = $LB("",530,"Abraham")
^MyApp.PersonD(2) = $LB("",680,"Philip")
```

Note that the first piece of the **$List** structure stored in each node is empty; this is reserved for a class name. If we define any subclasses of this Person class, this slot contains the subclass name. The **%OpenId** method (provided by the %Persistent class) uses this information to polymorphically open the correct type of object when multiple objects are stored within the same extent. This slot shows up in the class storage definition as a property named "%%CLASSNAME".

For more details, refer to the section on subclasses below.

CAUTION: Globals that are part of an extent are managed by corresponding ObjectScript and SQL code. Any changes made to such a global through direct global access may corrupt the structure of the global (rendering its data inaccessible) or otherwise compromise access to its data through ObjectScript or SQL.

To prevent this, you should not use the kill command on globals for tasks like dropping all the data in an extent. Instead, you should use API methods such as %KillExtent() or TRUNCATE TABLE, which perform important maintenance, such as resetting associated in-memory counters. However, classes that use a customized storage definition to project data from globals, are fully managed by application code, are exceptions to this rule. In such classes, you should consider setting either READONLY to 1 or MANAGEDEXTENT to 0 or both.

6.1.2 IDKEY

The IDKEY mechanism allows you to explicitly define the value used as an object ID. To do this, you simply add an IDKEY index definition to your class and specify the property or properties that will provide the ID value. Note that once you save an object, its object ID value cannot change. This means that after you save an object that uses the IDKEY mechanism, you can no longer modify any of the properties on which the object ID is based.

For example, we can modify the Person class used in the previous example to use an IDKEY index:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On Name [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save two instances of the Person class, the resulting global is now similar to:

```
^MyApp.PersonD("Abraham") = $LB("",530,"Abraham")
^MyApp.PersonD("Philip") = $LB("",680,"Philip")
```

Note that there is no longer any counter node defined. Also note that by basing the object ID on the Name property, we have implied that the value of Name must be unique for each object.

If the IDKEY index is based on multiple properties, then the main data nodes has multiple subscripts. For example:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On (Name,Age) [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

In this case, the resulting global will now be similar to:

```
 ^MyApp.PersonD("Abraham",530) = $LB("",530,"Abraham")
 ^MyApp.PersonD("Philip",680) = $LB("",680,"Philip")
```

Important: There must not be a sequential pair of vertical bars (||) within the values of any property used by an IDKEY index, unless that property is a valid reference to an instance of a persistent class. This restriction is imposed by the way in which the InterSystems SQL mechanism works. The use of || in IDKey properties can result in unpredictable behavior.

6.1.3 Subclasses

By default, any fields introduced by a subclass of a persistent object are stored in an additional node. The name of the subclass is used as an additional subscript value.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;

Property Age As %Integer;
}
```

Now we define a persistent subclass, MyApp.Student, that introduces two additional literal properties:

Class Definition

```
Class MyApp.Student Extends Person
{
Property Major As %String;

Property GPA As %Double;
}
```

If we create and save two instances of this MyApp.Student class, the resulting global will be similar to:

```
^MyApp.PersonD = 2  // counter node
^MyApp.PersonD(1) = $LB("Student",19,"Jack")
^MyApp.PersonD(1,"Student") = $LB(3.2,"Physics")

^MyApp.PersonD(2) = $LB("Student",20,"Jill")
^MyApp.PersonD(2,"Student") = $LB(3.8,"Chemistry")
```

The properties inherited from the Person class are stored in the main node, and those introduced by the Student class are stored in an additional subnode. This structure ensures that the Student data can be used interchangeably as Person data. For example, an SQL query listing names of all Person objects correctly picks up both Person and Student data. This structure also makes it easier for the Class Compiler to maintain data compatibility as properties are added to either the super- or subclasses.

Note that the first piece of the main node contains the string "Student" — this identifies nodes containing Student data.

6.1.4 Parent-Child Relationships

Within parent-child relationships, instances of child objects are stored as subnodes of the parent object to which they belong. This structure ensures that child instance data is physically clustered along with parent data.

For example, here is the definition for two related classes, Invoice:

Class Definition

```
/// An Invoice class
Class MyApp.Invoice Extends %Persistent
{
Property CustomerName As %String;

/// an Invoice has CHILDREN that are LineItems
Relationship Items As LineItem  [inverse = TheInvoice, cardinality = CHILDREN];
}
```

and LineItem:

Class Definition

```
/// A LineItem class
Class MyApp.LineItem Extends %Persistent
{
Property Product As %String;
Property Quantity As %Integer;

/// a LineItem has a PARENT that is an Invoice
Relationship TheInvoice As Invoice [inverse = Items, cardinality = PARENT];
}
```

If we store several instances of Invoice object, each with associated LineItem objects, the resulting global will be similar to:

```
^MyApp.InvoiceD = 2  // invoice counter node
^MyApp.InvoiceD(1) = $LB("","Wiley Coyote")
^MyApp.InvoiceD(1,"Items",1) = $LB("","Rocket Roller Skates",2)
^MyApp.InvoiceD(1,"Items",2) = $LB("","Acme Magnet",1)

^MyApp.InvoiceD(2) = $LB("","Road Runner")
^MyApp.InvoiceD(2,"Items",1) = $LB("","Birdseed",30)
```

For more information on relationships, see Relationships.

6.1.5 Embedded Objects

Embedded objects are stored by first converting them to a serialized state (by default a **$List** structure containing the object's properties) and then storing this serial state in the same way as any other property.

For example, suppose we define a simple serial (embeddable) class with two literal properties:

Class Definition

```
Class MyApp.MyAddress Extends %SerialObject
{
Property City As %String;
Property State As %String;
}
```

We now modify our earlier example to add an embedded Home address property:

Class Definition

```
Class MyApp.MyClass Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
Property Home As MyAddress;
}
```

If we create and save two instances of this class, the resulting global is equivalent to:

```
 ^MyApp.MyClassD = 2  // counter node
 ^MyApp.MyClassD(1) = $LB(530,"Abraham",$LB("UR","Mesopotamia"))
 ^MyApp.MyClassD(2) = $LB(680,"Philip",$LB("Bethsaida","Israel"))
```

6.1.6 Streams

Global streams are stored within globals by splitting their data into a series of chunks, each smaller than 32K bytes, and writing the chunks into a series of sequential nodes. File streams are stored in external files.

6.2 Indices

Persistent classes can define one or more indexes; additional data structures are used to make operations (such as sorting or conditional searches) more efficient. InterSystems SQL makes use of such indexes when executing queries. InterSystems IRIS Object and SQL automatically maintain the correct values within indexes as insert, update, and delete operations are carried out.

6.2.1 Storage Structure of Standard Indexes

A standard index associates an ordered set of one or more property values with the object ID values of the object containing the properties.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties and an index on its Name property:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Index NameIdx On Name;

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save several instances of this Person class, the resulting data and index globals is similar to:

```
 // data global
 ^MyApp.PersonD = 3  // counter node
 ^MyApp.PersonD(1) = $LB("",34,"Jones")
 ^MyApp.PersonD(2) = $LB("",22,"Smith")
 ^MyApp.PersonD(3) = $LB("",45,"Jones")


 // index global
 ^MyApp.PersonI("NameIdx"," JONES",1) = ""
 ^MyApp.PersonI("NameIdx"," JONES",3) = ""
 ^MyApp.PersonI("NameIdx"," SMITH",2) = ""
```

Note the following things about the index global:

1. By default, it is placed in a global whose name is the class name with an "I" (for Index) appended to it.
2. By default, the first subscript is the index name; this allows multiple indexes to be stored in the same global without conflict.
3. The second subscript contains the *collated* data value. In this case, the data is collated using the default SQLUPPER collation function. This converts all characters to uppercase (to sort without regard to case) and prepends a space character (to force all data to collate as strings).
4. The third subscript contains the Object ID value of the object that contains the indexed data value.
5. The nodes themselves are empty; all the needed data is held within the subscripts. Note that if an index definition specifies that data should be stored along with the index, it is placed in the nodes of the index global.

This index contains enough information to satisfy a number of queries, such as listing all Person class order by Name.

6.3 Bitmap Indexes

A bitmap index is similar to a standard index except that it uses a series of bitstrings to store the set of object ID values that correspond to the indexed value.

6.3.1 Logical Operation of Bitmap Indexes

A bitstring is a string containing a set of bits (0 and 1 values) in a special compressed format. InterSystems IRIS includes a set of functions to efficiently create and work with bitstrings:

- $Bit — Set or get a bit within a bitstring. ‘
- $BitCount — Count the number of bits within a bitstring.
- $BitFind — Find the next occurrence of a bit within a bitstring.
- $BitLogic — Perform logical (AND, OR) operations on two or more bitstrings.

Within a bitmap index, ordinal positions within a bitstring correspond to rows (Object ID number) within the indexed table. For a given value, a bitmap index maintains a bitstring that contains 1 for each row in which the given value is present, and contains 0 for every row in which it is absent. Note that bitmap indexes only work for objects that use the default storage structure with system-assigned, numeric Object ID values.

For example, suppose we have a table similar to the following:

ID	State	Product
1	MA	Hat
2	NY	Hat
3	NY	Chair
4	MA	Chair
5	MA	Hat

If the State and Product columns have bitmap indexes, then they contain the following values:

A bitmap index on the State column contains the following bitstring values:

MA	1	0	0	1	1
NY	0	1	1	0	0

Note that for the value, "MA", there is a 1 in the positions (1, 4, and 5) that correspond to the table rows with State equal to "MA".

Similarly, a bitmap index on the Product column contains the following bitstring values (note that the values are collated to uppercase within the index):

CHAIR	0	0	1	1	0
HAT	1	1	0	0	1

The InterSystems SQL Engine can execute a number of operations by iterating over, counting the bits within, or performing logical combinations (AND, OR) on the bitstrings maintained by these indexes. For example, to find all rows that have State equal to "MA" and Product equal to "HAT", the SQL Engine can simply combine the appropriate bitstrings together with logical AND.

In addition to these indexes, the system maintains an additional index, called an "extent index," that contains a 1 for every row that exists and a 0 for rows that do not (such as deleted rows). This is used for certain operations, such as negation.

6.3.2 Storage Structure of Bitmap Indexes

A bitmap index associates an ordered set of one or more property values with one or more bitstrings containing the Object ID values corresponding to the property values.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties and a bitmap index on its Age property:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Index AgeIdx On Age [Type = bitmap];

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save several instances of this Person class, the resulting data and index globals is similar to:

```
// data global
^MyApp.PersonD = 3  // counter node
^MyApp.PersonD(1) = $LB("",34,"Jones")
^MyApp.PersonD(2) = $LB("",34,"Smith")
^MyApp.PersonD(3) = $LB("",45,"Jones")

// index global
^MyApp.PersonI("AgeIdx",34,1) = 110...
^MyApp.PersonI("AgeIdx",45,1) = 001...

// extent index global
^MyApp.PersonI("$Person",1) = 111...
^MyApp.PersonI("$Person",2) = 111...
```

Note the following things about the index global:

1. By default, it is placed in a global whose name is the class name with an "I" (for Index) appended to it.
2. By default, the first subscript is the index name; this allows multiple indexes to be stored in the same global without conflict.

3. The second subscript contains the *collated* data value. In this case, a collation function is not applied as this is an index on numeric data.
4. The third subscript contains a *chunk* number; for efficiency, bitmap indexes are divided into a series of bitstrings each containing information for about 64000 rows from the table. Each of these bitstrings are referred to as a chunk.
5. The nodes contain the bitstrings.

Also note: because this table has a bitmap index, an extent index is automatically maintained. This extent index is stored within the index global and uses the class name, with a "$" character prepended to it, as its first subscript.

6.3.3 Direct Access of Bitmap Indexes

The following example uses a class extent index to compute the total number of stored object instances (rows). Note that it uses **$Order** to iterate over the chunks of the extent index (each chunk contains information for about 64000 rows):

Class Member

```
/// Return the number of objects for this class.<BR>
/// Equivalent to SELECT COUNT(*) FROM Person
ClassMethod Count() As %Integer
{
    New total,chunk,data
    Set total = 0

    Set chunk = $Order(^MyApp.PersonI("$Person",""),1,data)
    While (chunk '= "") {
        Set total = total + $bitcount(data,1)
        Set chunk = $Order(^MyApp.PersonI("$Person",chunk),1,data)
    }

    Quit total
}
```

7

Temporary Globals and the IRISTEMP Database

For some operations, you may need the power of globals without requiring the data to be saved indefinitely. For example, you may want to use a global to sort some data which you do not need to store to disk. For these operations, InterSystems IRIS® data platform provides the mechanism of *temporary globals*.

Temporary globals have the following characteristics:

- Temporary globals are stored within the IRISTEMP database, which is always defined to be a local (that is, a non-network) database. All globals mapped to the IRISTEMP database are treated as temporary globals.
- Changes to temporary globals are not written to disk. Instead the changes are maintained within the in-memory buffer pool. A large temporary global may be written to disk if there is not sufficient space for it within the buffer pool.
- For maximum efficiency, changes to temporary globals are not logged to a journal file.
- Temporary globals are automatically deleted whenever InterSystems IRIS is restarted. (Note: it can be a very long time before a live system is restarted; so you should not count on this for cleaning up temporary globals.)

Tip: Temporary globals are useful when you need temporary data for use by multiple processes. If you need temporary data for use only within a single process, consider using a process-private global, which is a special form of variable that is available only within the process that creates it and that is automatically removed when the process ends.

7.1 Using Temporary Globals

The mechanism for using temporary globals works as follows:

- For your application namespace, you define a global mapping so that globals with a specific naming convention are to be mapped to the IRISTEMP database, which is a special database as discussed below.

  For example, you might define a global mapping so that all globals with names of the form `^AcmeTemp*` are mapped to the IRISTEMP database.

- When your code needs to store data temporarily and read it again, your code writes to and reads from globals that use that naming convention.

  For example, to save a value, your code might do this:

```
set ^AcmeTempOrderApp("sortedarray")=some value
```

Then later your code might do this:

```
set somevariable = ^AcmeTempOrderApp("sortedarray")
```

By using temporary globals, you take advantage of the fact that the IRISTEMP database is not journaled. Because the database is not journaled, operations that use the database do not result in journal files. Journal files can become large and can cause space issues. However, note the following points:

- You cannot roll back any transactions that modify globals in the IRISTEMP database; this behavior is specific to IRISTEMP. If you need to manage temporary work via transactions, do not use globals in IRISTEMP for that purpose.
- Take care to use IRISTEMP only for work that does not need to be saved.
- The IRISTEMP database increases in size when it requires more memory. You can use the MaxIRISTempSizeAtStart parameter to help manage the size of IRISTEMP.

7.2 Defining a Mapping for Temporary Globals

To define a mapping for temporary globals, do the following:

1. Choose a naming convention and ensure that all of your developers are aware of it. Note the following points:
   - Consider whether to have many temporary globals or fewer temporary globals with multiple nodes. It is easier for InterSystems IRIS to efficiently read or write different nodes within the same global, compared to reading or writing the equivalent number of separate globals. The efficiency difference is negligible for small numbers of globals but is noticeable when there are hundreds of separate globals.
   - If you plan to use the same global mapping in multiple namespaces, then devise a system so that work in one namespace does not interfere with work in another namespace. For example, you could use the namespace name as a subscript in the globals.
   - Similarly, even within one namespace, devise a system so that each part of the code uses a different global or a different subscript in the same global, again to avoid interference.
   - Do not use system-reserved global names. See Global Variable Names to Avoid.
2. In the Management Portal, navigate to the **Namespaces** page (**System Administration** > **Configuration** > **System Configuration** > **Namespaces**).
3. In the row for your application namespace, click **Global Mappings**.
4. From the **Global Mappings** page, click **New Global Mapping** .
5. For **Global database location**, select IRISTEMP.
6. For **Global name**, enter a name ending in an asterisk (`*`). Do not include the initial caret of the name.

   For example: `AcmeTemp*`

   This mapping causes all globals with names that start `AcmeTemp*` to be mapped to the IRISTEMP database.
7. Click **OK**.

   Note: The **>>** symbol displayed in the first column of the new mappings row indicates that you opened the mapping for editing.
8. To save the mappings so that InterSystems IRIS uses them, click **Save Changes**.

For more details, see Configuring Namespaces.

7.3 System Use of IRISTEMP

Note that InterSystems uses temporary system globals as scratch space, for example, as temporary indexes during the execution of certain queries (for sorting, grouping, calculating aggregates, etc.). These globals are automatically mapped to IRISTEMP and include:

- ^*IRIS.Temp**
- ^*CacheTemp**
- ^*mtemp**

Never change any of these globals.

7.4 ^CacheTemp Globals

Historically, customers have used globals having names starting with `^CacheTemp` as temporary globals. By convention, these globals use names starting with `^CacheTempUser` to avoid possible conflict with temporary system globals. However, the best practice is to define your own temporary globals and map them to IRISTEMP, as described in Using Temporary Globals.

8

Management Portal Options

The Management Portal provides tools for viewing, modifying, and working with globals. This topic describes how to use these tools. Also see APIs.

For information on defining global mappings, see Configuring Namespaces.

8.1 General Advice

As with the ObjectScript commands **SET**, **MERGE**, **KILL**, and others, the tools described here provide direct access to manipulate globals. If you delete or modify via global access, you bypass all object and SQL integrity checking and there is no undo option. It is therefore important to be very careful when doing these tasks. (Viewing and exporting do not affect the database and are safe activities.)

CAUTION: Globals that are part of an extent are managed by corresponding ObjectScript and SQL code. Any changes made to such a global through direct global access may corrupt the structure of the global (rendering its data inaccessible) or otherwise compromise access to its data through ObjectScript or SQL.

To prevent this, you should avoid use the kill command on globals for tasks like dropping all the data in an extent. Instead, you should use API methods such as %KillExtent() or TRUNCATE TABLE, which perform important maintenance, such as resetting associated in-memory counters. However, classes that use a customized storage definition to project data from globals, are fully managed by application code, are exceptions to this rule. In such classes, you should consider setting either READONLY to 1 or MANAGEDEXTENT to 0 or both.

When using the tools described in this topic, make sure of the following:

- Be sure that you know which globals InterSystems IRIS® data platform uses. Not all of these are treated as "system" globals — that is, some of them are visible even when you do not select the **System** check box. Some of these globals store code, including your code.

  See Global Variable Names to Avoid.

- Be sure that you know which globals your application uses.

  Even if your application never performs any direct global access, your application uses globals. Remember that if you create persistent classes, their data and any indexes are stored in globals, whose names are based on the class names (by default). See Data.

8.2 Introduction to the Globals Page

The Management Portal includes the **Globals** page, which allows you to view, and edit globals in different ways. To access this page from the Management Portal home page:

1. Select **System Explorer** > **Globals**.
2. Select the namespace or database of interest:
   - Select either **Namespaces** or **Databases** from the **Lookin** list.
   - Select the desired namespace or database from the displayed list.

   Selecting a namespace or database updates the page to display its globals.
3. If you are looking for a particular global and do not initially see its name:
   - Optionally specify a search mask. To do so, enter a value into the **Globals** field. If you end the string with an asterisk "*", the asterisk is treated as a wildcard, and the page displays each global whose name begins with the string before the asterisk.

     After entering a value, press **Enter**.
   - Optionally select **System items** to include all system globals in the search.
   - Optionally select **Show SQL Table Names** to include **Table** and **Usage** columns in the globals table. If a global is used with an SQL table, these columns display the name of that table and its usage, such as whether it is a data/master map or a type of index.
   - Optionally select a value from **Page size**, which controls the number of globals to list on any page.

8.3 Viewing Global Data

The **View Global Data** page lists nodes of the given global. In the table, the first column displays the row number, the next column lists the nodes, and the right column shows the values. This page initially shows the first hundred nodes in the global.

To access this page, display the Globals page and select the **View** link next to the name of a global. Or click the **View** button.

On this page, you can do the following:

- Specify a search mask. To do so, edit the value in **Global Search Mask** as follows:
  - To display a single node, use a complete global reference. For example: `^Sample.PersonD(9)`
  - To display a subtree, use a partial global reference without the right parenthesis. For example: `^%SYS("JOURNAL"`
  - To display all nodes that match a given subscript, include the desired subscript and leave other subscript fields empty. For example: `^IRIS.Msg(,"en")`
  - To display all subtrees that match a given subscript, use a value as in the previous option but also omit the right parenthesis. For example: `^IRIS.Msg(,"en"`
  - To display nodes that match a range of subscripts, use *subscriptvalue1*`:`*subscriptvalue2* in the place of a subscript. For example: `^Sample.PersonD(50:60)`

    As with the previous option, if you omit the right parenthesis, the system displays the subtrees.

Then click **Display** or press **Enter**.

- Specify a different number of nodes to display. To do so, enter an integer into **Maximum Rows**.
- Repeat a previous search. To do so, select the search mask in the **Search History** drop-down.
- Select **Allow Edit** to make the data editable; see the next topic.

To close this page, click **Cancel**.

8.4 Editing Globals

CAUTION: Before making any edits, be sure that you know which globals InterSystems IRIS uses and which globals your application uses; see General Advice. There is no undo option. A modified global cannot be restored.

The **Edit Global Data** page enables you to edit globals. In the table, the first column displays the row number, the next column lists the nodes, and the right column shows the values (with a blue underline to indicate that the value can be edited). This page initially shows the first hundred nodes in the global.

To access and use this page:

1. Display the Globals page.
2. Select the **Edit** link next to the name of a global.
3. Optionally use the **Global Search Mask** field to refine what is displayed. See Viewing Global Data
4. Optionally specify a different number of nodes to display. To do so, enter an integer into **Maximum Rows**.
5. If necessary, navigate to the value you want to edit by selecting the subscripts that correspond to it.
6. Select the value that you want to edit.

   The page then displays two editable fields:

   - The top field contains the full global reference for the node you are editing. For example:
     `^Sample.PersonD("18")`

     You can edit this to refer to a different global node. If you do so, your action affects the newly specified global node.
   - The bottom field contains the current value of this node. For example:

     ```
     $lb("",43144,$lb("White","Orange"),$lb("8262 Elm Avenue","Islip","RI",57581),"Rogers,Emilio
     L.",
     $lb("7430 Washington Street","Albany","GA",66833),"650-37-4263","")
     ```

   Edit the values as needed.
7. If you make edits, click **Save** to save your changes, or click **Cancel**.

Or, to delete a node:

1. Optionally select **Delete global subnodes during deletion**
2. Click **Delete**.
3. Click **OK** to confirm this action.

Also see Performing Wholesale Replacements.

8.5 Exporting Globals

CAUTION: Because of how easy it is to import globals (which is an irreversible change), it is good practice to export only the globals you need to import. Note that if you export all globals, the export includes all the globals that contain code. Be sure that you know which globals InterSystems IRIS uses and which globals your application uses; see General Advice.

The **Export Globals** page enables you to export globals.

To access and use this page:

1. Display the Globals page.
2. Specify the globals to work with. To do so, see steps 2 and 3 in Introduction to the Globals Page.
3. Click the **Export** button.
4. Specify the file into which you wish to export the globals. To do this, either enter a file name (including its absolute or relative pathname) in the **Enter the path and name of the export on server <hostname>** field or click **Browse** and navigate to the file.
5. Select the export file's character set with the **Character set** list.
6. In the page's central box:
   - Choose an **Output format**
   - Choose a **Record format**
7. Select or clear **Check here to run export in the background...**
8. Click **Export**.
9. If the file already exists, click **OK** to overwrite it with a new version.

The export creates a .gof file.

8.6 Importing Globals

CAUTION: Before importing any globals, be sure that you know which globals InterSystems IRIS uses and which globals your application uses; see General Advice. There is no undo option. After you import a global into an existing global (thus merging the data), there is no way to restore the global to its previous state.

The **Import Globals** page enables you to import globals. To access and use this page:

1. Display the Globals page.
2. Click the **Import** button.
3. Specify the import file. To do this, either enter a file (including its absolute or relative pathname) in the **Enter the path and name of the import file** field or click **Browse** and navigate to the file.
4. Select the import file's character set with the **Character set** list.
5. Select **Next**.
6. Choose the globals to import using the check boxes in the table.

7. Optionally select **Run import in the background**. If you select this, the task is run in the background.
8. Click **Import**.

8.7 Finding Values in Globals

The **Find Global String** page enables you to find a given string in the subscripts or in the values of selected globals.

To access and use this page:

1. Display the Globals page.
2. Select the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."
3. Click the **Find** button.
4. For **Find What**, enter the string to search for.
5. Optionally clear **Match Case**. By default, the search is case-sensitive.
6. Click either **Find First** or **Find All**.

   The page then displays either the first node or all nodes whose subscripts or values contain the given string, within the selected globals. The table shows the node subscripts on the left and the corresponding values on the right.
7. If you used **Find First**, click **Find Next** to see the next node, as needed.
8. When you are done, click **Close Window**.

8.7.1 Performing Wholesale Replacements

CAUTION: Before making any edits, be sure that you know which globals InterSystems IRIS uses and which globals your application uses; see "General Advice." This option changes the data permanently. It is not recommended for use in production systems.

For development purposes, the **Find Global String** page also provides an option to make wholesale changes to values in global nodes. To use this option:

1. Display the Globals page.
2. Select the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."
3. Click the **Replace** button.
4. Use this page to find values as described in the previous section.
5. Specify a value for **Replace With**.
6. Click **Replace All**.
7. Click **OK** to confirm this action.

   The page then displays a preview of the change.
8. If the results are acceptable, click **Save**.
9. Click **OK** to confirm this action.

8.8 Deleting Globals

CAUTION: Before deleting any globals, be sure that you know which globals InterSystems IRIS uses and which globals your application uses; see General Advice. There is no undo option. A deleted global cannot be restored.

The **Delete Globals** page enables you to delete globals. To access and use this page:

1. Display the Globals page.
2. Select the globals to work with. To do so, see steps 2 and 3 in Introduction to the Globals Page.
3. Click the **Delete** button.
4. Click **OK** to confirm this action.

9

APIs for Working with Globals

InterSystems IRIS® data platform provides the following APIs to work with globals:

- The class %SYSTEM.OBJ provides the following methods:
  - **Export()** enables you to export globals to an XML file.
  - **Load()** and **LoadDir()** enable you to import globals contained in XML files.

  These are both available via the **$SYSTEM** variable, for example: **$SYSTEM.OBJ.Export**
- The class %Library.Global provides the following methods:
  - **Export()** enables you to export globals to .gof and other file formats (not including XML).
  - **Import()** enables you to import globals to .gof and other file formats (not including XML).

  %Library.Global also provides the **Get()** class query, which you can use to find globals, given search criteria.

For pointers to additional APIs, see Globals in the *InterSystems Programming Tools Index*.