
# Using the InterSystems Native API for .NET

Version 2019.1
2019-04-09

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

*Using the InterSystems Native API for .NET*
InterSystems IRIS Data Platform Version 2019.1 2019-04-09






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

# About This Book

InterSystems IRIS™ provides the lightweight *Native API for .NET* for direct access to the native data structure of InterSystems databases. *Globals* are tree-based sparse arrays used to implement the InterSystems multidimensional storage model. These native data structures provide very fast, flexible storage and retrieval. InterSystems IRIS uses globals to make data available as objects or relational tables, but you can use the Native API to implement your own data structures.

The following chapters discuss the main features of the Native API:

- Introduction to the Native API — demonstrates how to create an instance of the main Native API class, open a connection, and perform some simple database operations.
- Working with Global Arrays — describes how to create, change, or delete nodes in a multidimensional global array, and demonstrates methods for iteration, transactions, and locking.
- Calling ObjectScript Methods and Functions — describes a set of methods that allow an application to call user defined ObjectScript class methods and functions on the server.
- Native API Quick Reference — provides a brief description of each method in the Native API.

There is also a detailed Table of Contents.

### More information about globals

A version of the Native API is also available for Java:

- *Using the InterSystems Native API for Java*

The following book is highly recommended for developers who want to master the full power of globals:

- *Using Globals* — describes how to use globals in ObjectScript, and provides more information about how multidimensional storage is implemented on the server.

### InterSystems Core APIs for .NET

The Native API is part of a suite that also includes lightweight APIs for object and relational database access. See the following books for more information:

- *Using the InterSystems Managed Provider for .NET* — describes how use the InterSystems implementation of the ADO.NET Managed Provider, which provides data access through both ADO .NET data providers and the Entity Framework.
- *Persisting .NET Objects with XEP* — describes how to store and retrieve .NET objects using the InterSystems event persistence API (XEP).

# 1

# Introduction to the Native API

The *Native API for .NET* is a lightweight interface to the native multidimensional storage data structures that underlie the InterSystems IRIS™ object and SQL interfaces. The Native API allows you to implement your own data structures by providing direct access to *global arrays*, the tree-based sparse arrays that form the basis of the multidimensional storage model. The Native API for .NET is implemented in the IrisClient.ADO.IRIS class as an extension to the InterSystems Managed Provider for .NET (see *Using the InterSystems Managed Provider for .NET*).

This chapter discusses the following topics:

- Introduction to Global Arrays — introduces global array concepts and provides a simple demonstration of how the Native API is used.
- Glossary of Native API Terms — defines some important terms used in this book.

## 1.1 Introduction to Global Arrays

A global array, like all sparse arrays, is a tree structure rather than a sequentially numbered list. The basic concept behind global arrays can be illustrated by analogy to a file structure. Each *directory* in the tree is uniquely identified by a *path* composed of a *root directory* identifier followed by a series of *subdirectory* identifiers, and any directory may or may not contain *data*.

Global arrays work the same way: each *node* in the tree is uniquely identified by a *node address* composed of a *global name* identifier and a series of *subscript* identifiers, and a node may or may not contain a *value*. For example, here is a global array consisting of five nodes, two of which contain values:

```
root -->|--> foo --> SubFoo="A"
        |--> bar --> lowbar --> UnderBar=123
```

Values could be stored in the other possible node addresses (for example, root or root->bar), but no resources are wasted if those node addresses are *valueless*. In InterSystems globals notation, the two nodes with values would be:

```
^root("foo","SubFoo")
^root("bar","lowbar","UnderBar")
```

In this notation, the circumflex (^) is a symbol indicating that this is a global array. This is followed by the global name (`root`) and a comma-delimited *subscript list* in parentheses, specifying the entire path to the node. This global array could be created by two calls to the Native API **Set()** method:

```
native.Set("A","root","foo","SubFoo");
native.Set(123,"root","bar","lowbar","UnderBar");
```

Global array `root` is created when the first call assigns value `"A"` to node *^root("foo","SubFoo")*. Nodes can be created in any order, and with any set of subscripts. The same global array would be created if we reversed the order of these two calls. The valueless nodes are created automatically (and will be deleted automatically when no longer needed. For details, see "Creating, Updating, and Deleting Nodes" in the next chapter).

The Native API code to create this array is demonstrated in the following example. An IRISConnection object connects to the server and creates an instance of class IRIS, which implements the Native API. IRIS methods create a global array, read the resulting persistent values from the database, and then delete the global array.

### The NativeDemo Program

The Native API for .NET is part of the InterSystems.Data.IrisClient.dll library. For detailed information on installation and usage, see the Introduction to *Using the InterSystems Managed Provider for .NET*.

```
using System;
using InterSystems.Data.IRISClient;
using InterSystems.Data.IRISClient.ADO;

namespace NativeSpace {
  class NativeDemo {
    static void Main(string[] args) {
      try {

//Open a connection to the server and create an IRIS object
        IRISConnection conn = new IRISConnection();
        conn.ConnectionString = "Server = localhost; "
        + "Port = 51773; " + "Namespace = USER; "
        + "Password = SYS; " + "User ID = _SYSTEM;";
        conn.Open();
        IRIS native = IRIS.CreateIRIS(conn);

//Create a global array in the USER namespace on the server
        native.Set("A", "root", "foo", "SubFoo");
        native.Set(123, "root", "bar", "lowbar", "UnderBar");

// Read the values from the database and print them
        string subfoo = native.GetString("root", "foo", "SubFoo");
        string underbar = native.GetString("root", "bar", "lowbar", "UnderBar");
        Console.WriteLine("Created two values: \n"
          + " ^root(\"foo\",\"SubFoo\")=" + subfoo + "\n"
          + " ^root(\"bar\",\"lowbar\",\"UnderBar\")=" + underbar);

//Delete the global array and terminate
        native.Kill("root"); // delete global array ^root
        native.Close();
        conn.Close();
      }
      catch (Exception e) {
        Console.WriteLine(e.Message);
      }
    }// end Main()
  } // end class NativeDemo
}
```

NativeDemo prints the following lines:

```
Created two values:
   ^root("foo","SubFoo")=A
   ^root("bar","lowbar","UnderBar")=123
```

In this example, an IRISConnection object named *conn* provides a connection to the database associated with the USER namespace. Native API methods perform the following actions:

- IRIS.**CreateIRIS()** creates a new instance of IRIS named *native*, which will access the database through *conn*.
- IRIS.**Set()** creates new persistent nodes in the database.
- IRIS.**GetString()** queries the database and returns the values of the specified nodes.
- IRIS.**Kill()** deletes the specified node and all of its subnodes from the database.

The next chapter provides detailed explanations and examples for all of these methods.

# 1.2 Glossary of Native API Terms

See earlier sections of this chapter for an overview of the concepts listed here. Examples in this glossary will refer to the global array structure listed below. The *^Legs* global array has ten nodes and three node levels. Seven of the ten nodes contain values:

```
Legs                      // root node, valueless, 3 child nodes
  fish = 0                // level 1 node, value=0
  mammal                  // level 1 node, valueless
    human = 2             // level 2 node, value=2
    dog = 4               // level 2 node, value=4
  bug                     // level 1 node, valueless, 3 child nodes
    insect = 6            // level 2 node, value=6
    spider = 8            // level 2 node, value=8
    millipede = Diplopoda // level 2 node, value="Diplopoda", 1 child node
      centipede = 100     // level 3 node, value=100
```

### Child node

The nodes immediately under a given parent node. The address of a child node is specified by adding exactly one subscript to the end of the parent subscript list. For example, parent node *^Legs("mammal")* has child nodes *^Legs("mammal","human")* and *^Legs("mammal","dog")*.

### Global name

The identifier for the root node is also the name of the entire global array. For example, root node identifier `Legs` is the global name of global array *^Legs*. The circumflex ^ is not part of the identifier; it is an ObjectScript convention to indicate that we are talking about a node address.

### Node

An element of a global array, uniquely identified by a namespace consisting of a global name and an arbitrary number of subscript identifiers. A node must either contain data, have child nodes, or both.

### Node level

The number of subscripts in the node address. A ‘level 2 node’ is just another way of saying ‘a node with two subscripts’. For example, *^Legs("mammal","dog")* is a level 2 node. It is two levels under root node *^Legs* and one level under *^Legs("mammal")*.

### Node address

The complete namespace of a node, including the global name and all subscripts. In ObjectScript notation, a node address is written as a circumflex (^) followed by the root node identifier and an optional list of subscripts in parentheses. For example, node address *^Legs(fish)* consists of root node identifier `Legs` plus a list containing one subscript, `fish`. Depending on context, *^Legs* (with no subscript list) can refer to either the root node address or the entire global array.

### Root node

The node with no subscripts at the base of the global array tree (see Global name).

### Subnode

All descendants of a given node are referred to as *subnodes* of that node. For example, node *^Legs("bug")* has four different subnodes on two levels. All nine subscripted nodes are subnodes of root node *^Legs*.

### Subscript / Subscript list

All nodes under the root node are addressed by specifying the global name and a list of one or more subscript identifiers (see Node address).

### Target address

Many Native API methods require you to specify a valid node address that does not necessarily point to an existing node. For example, the **set()** method takes a *value* argument and a target address, and stores the value at that address. If no node exists at the target address, a new node is created.

### Value

A node can contain a value of any supported type. A node with no child nodes must contain a value; otherwise the value is optional.

### Valueless node

A node must either contain data, have child nodes, or both. A node that has child nodes but does not contain data is called a valueless node.

# 2

# Working with Global Arrays

This chapter covers the following topics:

- Creating, Updating, and Deleting Nodes — describes how to create a global array and set or change node values.
- Finding Nodes in a Global Array — describes iteration methods that allow rapid access to the nodes of a global array.
- Transactions and Locking — describes how to use the Native API in transactions.

## 2.1 Creating, Updating, and Deleting Nodes

This section describes the Native API methods used to create, update, and delete nodes. **Set()**, **Increment()**, and **Kill()** are the only methods that can create a global array or alter its contents. The following examples demonstrate how to use each of these methods.

### Setting and changing node values

IRIS.**Set()** takes a *value* argument and stores the value at the specified address. If no node exists at that address, a new one is created.

The **Set()** method can assign values of any supported datatype. In the following example, the first call to **Set()** creates a new node at subnode address *^myGlobal("A")* and sets the value of the node to string `"first"`. The second call changes the value of the subnode, replacing it with integer `1`.

```
native.Set("first", "myGlobal", "A");     // create node ^myGlobal("A") = "first"
native.Set(1, "myGlobal", "A");   // change value of ^myGlobal("A") to 1.
```

**Set()** is a polymorphic accessor that can create and change values of any supported datatype, as demonstrated in this example. To read an existing value, you must use a different **Get<Type>()** accessor method for each datatype.

### Incrementing node values

IRIS.**Increment()** takes an integer *number* argument, increments the node value by that amount, and returns the incremented value. It uses a thread-safe atomic operation to change the value of the node, so the node is never locked. If there is no node at the target address, the method creates one and assigns the *number* argument as the value. In the following example, the first call to **Increment()** creates new subnode *^myGlobal("B")* with value `-2`. The next two calls each increment by `-2`, resulting in a final value of `-6`:

```
for (int loop = 0; loop < 3; loop++) {
  native.Increment(-2,"myGlobal", "B");
}
```

**Note:** ***Naming rules***

Global names and subscripts obey the following rules:

- The length of a *node address* (totaling the length of the global name and all subscripts) can be up to 511 characters. (Some typed characters may count as more than one encoded character for this limit. For more information, see "Maximum Length of a Global Reference" in *Using Globals*).
- A *global name* can include letters, numbers, and periods (" . "), and can have a length of up to 31 significant characters. It must begin with a letter, and must not end with a period.
- A *subscript* can be a string, an integer, or a number. String subscripts are case-sensitive, and can include characters of all types. Length is limited only by the 511 character maximum for the total node address.

### Deleting a node or group of nodes

IRIS.**Kill()** — deletes the specified node and all of its subnodes. The entire global array will be deleted if the root node is deleted or if all nodes with values are deleted.

Global array *^myGlobal* initially contains the following nodes:

```
^myGlobal = <valueless node>
  ^myGlobal("A") = 0
    ^myGlobal("A",1) = 0
    ^myGlobal("A",2) = 0
  ^myGlobal("B") = <valueless node>
    ^myGlobal("B",1) = 0
```

This example will delete the global array by calling **Kill()** on two of its subnodes. The first call will delete *^myGlobal("A")* and both of its subnodes:

```
native.Kill("myGlobal", "A");
// also kills child nodes ^myGlobal("A",1) and ^myGlobal("A",2)
```

The second call deletes the last remaining subnode with a value, killing the entire global array:

```
native.Kill("myGlobal","B",1);
```

- The parent node, *^myGlobal("B")*, is deleted because it is valueless and now has no subnodes.
- Root node *^myGlobal* is valueless and now has no subnodes, so the entire global array is deleted from the database.

Of course, the global array could also have been deleted by calling **Kill()** on root node *^myGlobal*:

```
native.Kill("myGlobal");
```

# 2.2 Finding Nodes in a Global Array

The Native API provides ways to iterate over part or all of a global array. The following topics describe the various iteration methods:

- Iterating Over a Set of Child Nodes — describes how to iterate over all child nodes under a given parent node.
- Iteration in Conditional Loops — describes methods and properties that provide more control over iteration.
- Testing for Child Nodes and Node Values — describes how to find all subnodes regardless of node level, and identify which nodes have values.

## 2.2.1 Iterating Over a Set of Child Nodes

Child nodes are sets of nodes immediately under the same parent node. Any child node address can be defined by appending one subscript to the subscript list of the parent. For example, the following global array has four child nodes under parent node *^heros("dogs")*:

### The ^heros global array

This global array uses the names of several heroic dogs (plus a reckless boy and a pioneering sheep) as subscripts. The values are birth years.

```
^heros                                          // root node,    valueless, 2 child nodes
   ^heros("dogs")                               // level 1 node, valueless, 4 child nodes
      ^heros("dogs","Balto") = 1919             // level 2 node, value=1919
      ^heros("dogs","Hachiko") = 1923           // level 2 node, value=1923
      ^heros("dogs","Lassie") = 1940            // level 2 node, value=1940, 1 child node
         ^heros("dogs","Lassie","Timmy") = 1954 // level 3 node, value=1954
      ^heros("dogs","Whitefang") = 1906         // level 2 node, value=1906
   ^heros("sheep")                              // level 2 node, valueless, 1 child node
      ^heros("sheep","Dolly") = 1996            // level 2 node, value=1996
```

The following methods are used to create an iterator, define the direction of iteration, and set the starting point of the search:

- IRIS.**GetIRISIterator()** returns an instance of IRISIterator for the child nodes of the specified target node.
- IRIS.**GetIRISReverseIterator()** returns an instance of IRISIterator set to backward iteration for the child nodes of the specified target node.
- IRISIterator.**StartFrom()** sets the iterator's starting position to the specified subscript. The subscript is an arbitrary starting point, and does not have to address an existing node.

### Read child node values in reverse order

The following code iterates over child nodes of *^heros("dogs")* in reverse collation order, starting with subscript V:

```
// Create a reverse iterator for child nodes of ^heros("dogs")
    IRISIterator iterDogs = native.GetIRISReverseIterator("heros","dogs");
// Start the search with subscript "V" and iterate to lowest collation value
    iterDogs.StartFrom("V");

    Console.Write("Dog birth years: ");
    foreach (int BirthYear in iterDogs) {
        Console.Write(BirthYear + " ");
    };
```

This code prints the following output:

```
Dog birth years: 1940 1923 1919
```

The example does the following:

- **GetIRISReverseIterator()** returns iterator *iterDogs*, which will find child nodes of *^heros("dogs")* in reverse collation order.
- **StartFrom()** specifies subscript V, meaning that the search range will include all child nodes of *^heros("dogs")* with subscripts lower than V in collation order. The iterator will first find subscript `Lassie`, followed by `Hachiko` and `Balto`.

Two subnodes of *^heros("dogs")* are ignored:

- Child node *^heros("dogs","Whitefang")* will not be found because it is outside of the search range (`Whitefang` is higher than V in collation order).

- Level 3 node *^heros("dogs","Lassie","Timmy")* will not be found because it is a child of `Lassie`, not `dogs`.

See the last section in this chapter (“Testing for Child Nodes and Node Values”) for a discussion of how to iterate over multiple node levels.

**Note:** **Collation Order**

The order in which nodes are retrieved depends on the *collation order* of the subscripts. This is not a function of the iterator. When a node is created, it is automatically stored it in the collation order specified by the storage definition. In this example, the child nodes of *^heros("dogs")* would be stored in the order shown (`Balto`, `Hachiko`, `Lassie`, `Whitefang`) regardless of the order in which they were created. For more information, see “Collation of Global Nodes” in *Using Globals*.

## 2.2.2 Iteration in Conditional Loops

The previous section demonstrated an easy way to make a single pass over a set of child nodes, but in some cases you may want more control than a simple `foreach` loop can provide. This section demonstrates some methods and properties that allow more control over the iterator and provide easier access to data:

- IRISIterator.**MoveNext()** implements System.Collections.IEnumerator, allowing you to control exactly when the iterator will move to the next node. It returns `true` if the next node has been found, or `false` if there are no more nodes in the current iteration.
- IRISIterator.**Reset()** can be called after exiting a loop to reset the iterator to its starting position, allowing it to be used again.
- IRISIterator.**Current** gets an object containing the value of the node at the current iterator position. This is the same value as the one assigned to the current loop variable in a `foreach` loop.
- IRISIterator.**CurrentSubscript** gets an object containing the lowest level subscript for the node at the current iterator position. For example, if the iterator points to node *^myGlobal(23,"somenode")*, the returned object will contain value `"somenode"`.

Like the previous example, this one uses the ^heros global array and iterates over the child nodes under *^heros("dogs")*. However, this example uses the same iterator to make several passes over the child nodes, and exits a loop as soon as certain conditions are met.

### Search for values that match items in a list

This example scans the child nodes under *^heros("dogs")* until it finds a specific node value or runs out of nodes. Array *targetDates* specifies the list of *targetYear* values to be used in the main `foreach` loop. Within the main loop, the `do while` loop finds each child node and compares its value to the current *targetYear*.

```
IRISIterator iterDogs = native.GetIRISIterator("^heros","dogs");
bool seek;
int[] targetDates = {1906, 1940, 2001};
foreach (int targetYear in targetDates) {
  do {
    seek = iterDogs.MoveNext();
    if (!seek) {
      Console.WriteLine("Could not find a dog born in " + targetYear);
    }
    else if ((int)iterDogs.Current == targetYear) {
      Console.WriteLine(iterDogs.CurrentSubscript + " was born in " + iterDogs.Current);
      seek = false;
    }
  } while (seek);
  iterDogs.Reset();
} // end foreach
```

This code prints the following output:

```
Whitefang was born in 1906
Lassie was born in 1940
Could not find a dog born in 2001
```

The example does the following:

- **GetIRISIterator()** returns iterator *iterDogs*, which will find child nodes of *^heros("dogs")* in collation order (as demonstrated in the previous section, "Iterating Over a Set of Child Nodes"). *iterDogs* will be reset and used again in each pass of the `foreach` loop.
- **MoveNext()** is called in each pass of the `do while` loop to find the next child node. It sets *seek* to `true` if a node is found, or `false` if there are no more child nodes. If *seek* is `false`, the `do while` loop exits after printing a message indicating that the current *targetYear* value was not found.
- The **Current** and **CurrentSubscript** properties of *iterDogs* are set each time a child node is found. **Current** contains the current node value, and **CurrentSubscript** contains the current subscript.
- **Current** is compared to *targetYear*. If there is a match, a message displays both the subscript and the node value, and the `do while` loop is terminated by setting *seek* to `false`.
- **Reset()** is called at the end of each `do while` pass. This returns iterator *iterDogs* to its original starting condition so it can be used again in the next pass.

## 2.2.3 Testing for Child Nodes and Node Values

In the previous examples, the scope of the search is restricted to child nodes of *^heros("dogs")*. The iterator fails to find two values in global array ^heros because they are under different parents:

- Level 3 node *^heros("dogs","Lassie","Timmy")* will not be found because it is a child of `Lassie`, not `dogs`.
- Level 2 node *^heros("sheep","Dolly")* is not found because it is a child of `sheep`, not `dogs`.

To search the entire global array, we need to find all of the nodes that have child nodes, and create an iterator for each one. The **IsDefined()** method provides the necessary information:

- IRIS.**IsDefined()** — can be used to determine if a node has a value, a subnode, or both. It returns one of the following values:
  - `0` — the specified node does not exist
  - `1` — the node exists and has a value
  - `10` — the node is valueless but has a child node
  - `11` — the node has both a value and a child node

  The returned value can be used to determine several useful boolean values:

```
bool exists = (native.IsDefined(root,subscripts) > 0);      // value is 1, 10, or 11
bool hasValue = (native.IsDefined(root,subscripts)%10 > 0); // value is 1 or 11
bool hasChild = (native.IsDefined(root,subscripts) > 9);    // value is 10 or 11
```

The following example consists of two methods:

- **ProcessNode()** will be called for each node in the *^heros* global array. It calls **IsDefined()** on the current node, and returns a boolean value indicating whether the node has child nodes. It also checks to see if the current subscript is `Timmy` or `Dolly`, and prints a message if so.
- **FindLostHeros()** uses the return value of **ProcessNode()** to navigate the entire global array. It starts by iterating over the child nodes of root node *^heros*. Whenever **ProcessNode()** indicates that the current node has child nodes, **FindLostHeros()** creates a new iterator to test the lower level child nodes.

### Method ProcessNode()

```
public bool ProcessNode(IRISIterator iter, string root, params object[] subscripts) {
// Test for values and child nodes
  int state = native.IsDefined(root,subscripts);
  bool hasValue = (state%10 > 0); // has value if state is 1 or 11

// Look for lost heros
  string[] lost = {"Timmy","Dolly"};
  if (hasValue) { // ignore valueless nodes
    string name = (string)iter.CurrentSubscript;
    int year = (int)iter.Current;
    foreach (string hero in lost) {
      if (hero == name) {
        Console.WriteLine("Hey, we found " + name + " (born in " + year + ")!!!");
      }
    }
  }

  bool hasChild = (state > 9);    // has child if state is 10 or 11
  return hasChild;
}
```

### Method FindLostHeros()

This example processes a known structure, and traverses the various levels with simple nested calls. In the less common case where a structure has an arbitrary number of levels, a recursive algorithm could be used.

```
public void FindLostHeros() {
  string root = "heros";

// Iterate over child nodes of root node ^heros
  IRISIterator iterRoot = native.GetIRISIterator(root);
  foreach (object node in iterRoot) {
    object sub1 = iterRoot.CurrentSubscript;
    bool hasChild1 = ProcessNode(iterRoot,sub1);

    // Process current child of ^heros(sub1)
    if (hasChild1) {
      IRISIterator iterOne = native.GetIRISIterator(root,sub1);
      foreach (object node in iterOne) {
        object sub2 = iterOne.CurrentSubscript;
        bool hasChild2 = ProcessNode(iterOne,sub1,sub2);

      // Process current child of ^heros(sub1,sub2)
        if (hasChild2) {
          IRISIterator iterTwo = native.GetIRISIterator(root,sub1,sub2);
          foreach (object node in iterTwo) {
            object sub3 = iterTwo.CurrentSubscript;
            ProcessNode(iterTwo,sub1,sub2,sub3); //no child nodes below level 3
          }
        } //end hasChild2
      }
    } //end hasChild1
  } // end main loop
} // end FindLostHeros()
```

# 2.3 Transactions and Locking

The following topics are discussed in this section:

- Controlling Transactions — describes methods used to process transactions.
- Acquiring and Releasing Locks — describes how to use the various lock methods.
- Using Locks in a Transaction — provides examples of locking within a transaction.

## 2.3.1 Controlling Transactions

The Native API provides the following methods to control transactions:

- IRIS.**TCommit()** — commits one level of transaction.
- IRIS.**TStart()** — starts a transaction (which may be a nested transaction).
- IRIS.**GetTLevel()** — returns an int value indicating the current transaction level (0 if not in a transaction).
- IRIS.**TRollback()** — rolls back all open transactions in the session.
- IRIS.**TRollbackOne()** — rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back.

The following example starts three levels of nested transaction, setting the value of a different node in each transaction level. All three nodes are printed to prove that they have values. The example then rolls back the second and third levels and commits the first level. All three nodes are printed again to prove that only the first node still has a value.

### *Controlling Transactions*: Using three levels of nested transaction

```
        String globalName = "myGlobal";
        native.TStart();

      // GetTLevel() is 1: create ^myGlobal(1) = "firstValue"
        native.Set("firstValue", globalName, native.GetTLevel());

        native.TStart();
      // GetTLevel() is 2: create ^myGlobal(2) = "secondValue"
        native.Set("secondValue", globalName, native.GetTLevel());

        native.TStart();
      // GetTLevel() is 3: create ^myGlobal(3) = "thirdValue"
        native.Set("thirdValue", globalName, native.GetTLevel());

        Console.WriteLine("Node values before rollback and commit:");
        for (int ii=1;ii<4;ii++) {
          Console.Write(globalName + "(" + ii + ") = ");
         if (native.IsDefined(globalName,ii) > 1) Console.WriteLine(native.getString(globalName,ii));

          else Console.WriteLine("<valueless>");
        }
      // prints: Node values before rollback and commit:
      //         ^myGlobal(1) = firstValue
      //         ^myGlobal(2) = secondValue
      //         ^myGlobal(3) = thirdValue

        native.TRollbackOne();
        native.TRollbackOne();  // roll back 2 levels to getTLevel 1
        native.TCommit();  // GetTLevel() after commit will be 0
        Console.WriteLine("Node values after the transaction is committed:");
        for (int ii=1;ii<4;ii++) {
          Console.Write(globalName + "(" + ii + ") = ");
         if (native.IsDefined(globalName,ii) > 1) Console.WriteLine(native.getString(globalName,ii));

          else Console.WriteLine("<valueless>");
        }
      // prints: Node values after the transaction is committed:
      //         ^myGlobal(1) = firstValue
      //         ^myGlobal(2) = <valueless>
      //         ^myGlobal(3) = <valueless>
```

## 2.3.2 Acquiring and Releasing Locks

The following methods of class IRIS are used to acquire and release locks. Both methods take a *lockMode* argument to specify whether the lock is shared or exclusive:

```
lock (String lockMode, Integer timeout, String globalName, String...subscripts)  final boolean
unlock (String lockMode, String globalName, String...subscripts)  final void
```

- IRIS.**Lock()** — Takes *lockMode*, *timeout*, *globalName*, and *subscripts* arguments, and locks the node. The *lockMode* argument specifies whether any previously held locks should be released. This method will time out after a predefined interval if the lock cannot be acquired.
- IRIS.**Unlock()** — Takes *lockMode*, *globalName*, and *subscripts* arguments, and releases the lock on a node.

The following argument values can be used:

- *lockMode* — combination of the following chars, S for shared lock, E for escalating lock, default is empty string (exclusive and non-escalating)
- *timeout* — amount to wait to acquire the lock in seconds

**Note:** You can use the Management Portal to examine locks. Go to **[System Operation] > [Locks]** to see a list of the locked items on your system.

## 2.3.3 Using Locks in a Transaction

This section demonstrates incremental locking within a transaction, using the methods previously described (see "Controlling Transactions" and "Acquiring and Releasing Locks"). You can see a list of the locked items on your system by opening the Management Portal and going to **[System Operation] > [Locks]**. The calls to **ReadKey()** in the following code will pause execution so that you can look at the list whenever it changes.

There are two ways to release all currently held locks:

- IRIS.**ReleaseAllLocks()** — releases all locks currently held by this connection.
- When the **Close()** method of the connection object is called, it releases all locks and other connection resources.

The following examples demonstrate the various lock and release methods.

### Using incremental locking in transactions

```
  native.Set("my node", "nodeRef1", "my-node");
  native.Set("shared node", "nodeRef2", "shared-node");

  try {
    native.TStart();
// lock ^nodeRef1("my-node") exclusively
    native.Lock("E",10,"nodeRef1", "my-node");
// lock ^nodeRef2 shared
    native.Lock("ES",10,"nodeRef2", "shared-node");
    Console.WriteLine("Exclusive lock on ^nodeRef1(\"my-node\") and shared lock on ^nodeRef2");

    Console.WriteLine("Press return to release locks individually");
    Console.ReadKey(); // Wait for user to press Return

// release ^nodeRef1("my-node") after transaction
    native.Unlock("E",,"nodeRef1", "my-node");
// release ^nodeRef2 immediately
    native.Unlock("ES",,"nodeRef2", "shared-node");
    Console.WriteLine("Press return to commit transaction");
    Console.ReadKey();
    native.TCommit();
  }
  catch (Exception e) { Console.WriteLine(e.Message); }
  catch (System.IO.IOException e) { Console.WriteLine(e.Message); }
```

### Using non-incremental locking in transactions

```
// lock ^nodeRef1("my-node") non-incremental
  native.Lock("",10,"nodeRef1", "my-node");
  Console.WriteLine("Exclusive lock on ^nodeRef1(\"my-node\"), return to lock ^nodeRef1
non-incrementally");
  Console.ReadKey();

// lock ^nodeRef2 shared non-incremental
  native.Lock("S",10,"nodeRef2", "shared-node");
  Console.WriteLine("Verify that only ^nodeRef2 is now locked, then press return");
  Console.ReadKey();
```

### Using ReleaseAllLocks() in transactions to release all incremental locks

```
          // lock ^nodeRef1("my-node") shared incremental
          native.Lock("SE",10,"nodeRef1", "my-node");;

        // lock ^nodeRef2 exclusive incremental
          native.Lock("E",10,"nodeRef2", "shared-node");
          Console.WriteLine("Two locks are held (one with lock count 2), return to release both locks");

          Console.ReadKey();

          native.ReleaseAllLocks();
          Console.WriteLine("Verify both locks have been released");
          Console.ReadKey();
```

# 3

# Calling ObjectScript Methods and Functions

This chapter describes a set of IRIS class methods that allow an application to call ObjectScript class methods and functions from the InterSystems IRIS™ class library.

- Class Method Calls — demonstrates how to call ObjectScript class methods.
- Function Calls — demonstrates how to call ObjectScript functions and procedures.
- Sample ObjectScript Methods and Functions — lists the ObjectScript methods and functions used by these examples.

## 3.1 Class Method Calls

The following Native API methods call a specified ObjectScript class method. They take String arguments for *className* and *methodName*, plus an Object array containing `0` or more method arguments. The result is returned as the indicated type:

- **ClassMethodBool()**
- **ClassMethodBytes()**
- **ClassMethodDouble()**
- **ClassMethodLong()**
- **ClassMethodString()**
- **ClassMethodVoid()**

Trailing arguments may be omitted in argument lists, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. An exception will be thrown if a non-null argument is passed to the right of a null argument.

The code in this example calls class methods of each datatype from an ObjectScript test class named User.NativeTest (see "ObjectScript Class User.NativeTest" in the following section for details).

### Calling class methods from ObjectScript class User.NativeTest

```
String className = "User.NativeTest";

Console.Write("Calling cosBoolean class method. Result: ");
Console.WriteLine(IRIS.ClassMethodBool(className,"cosBoolean",false));
```

```
Console.Write("Calling cosBytes class method. Result: ");
Console.WriteLine(new String(IRIS.ClassMethodBytes(className,"cosBytes","byteArray")));

Console.Write("Calling cosString class method. Result: ");
Console.WriteLine(IRIS.ClassMethodString(className,"cosString","Java Test"));

Console.Write("Calling cosLong class method. Result: ");
Console.WriteLine(IRIS.ClassMethodLong(className,"cosLong",7,8));

Console.Write("Calling cosDouble class method. Result: ");
Console.WriteLine(IRIS.ClassMethodDouble(className,"cosDouble",7.56));

Console.Write("Calling cosVoid class method. Result: ");
IRIS.ClassMethodVoid(className,"cosVoid",67);
Console.WriteLine("(method returned successfully)");
```

# 3.2 Function Calls

The following methods call user-defined ObjectScript functions or procedures. They take String arguments for *functionName* and *routineName*, plus an Object array containing `0` or more function arguments. The result is returned as the indicated type:

- **FunctionBool()**
- **FunctionBytes()**
- **FunctionDouble()**
- **FunctionLong()**
- **FunctionString()**
- **Procedure()**

Trailing arguments may be omitted in argument lists, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. An exception will be thrown if a non-null argument is passed to the right of a null argument.

The code in this example calls functions of each datatype from an ObjectScript test routine named NativeRoutine (File NativeRoutine.mac. See “ObjectScript Routine NativeRoutine.mac” in the following section for details).

### Calling functions from NativeRoutine.mac

```
String routineName = "NativeRoutine";

Console.Write("Calling funcBoolean function. Result: ");
Console.WriteLine(IRIS.FunctionBool("funcBoolean",routineName,false));

Console.Write("Calling funcBytes function. Result: ");
Console.WriteLine(new String(IRIS.FunctionBytes ("funcBytes",routineName,"byteArray")));

Console.Write("Calling funcString function. Result: ");
Console.WriteLine(IRIS.FunctionString ("funcString",routineName,"Test String"));

Console.Write("Calling funcLong function. Result: ");
Console.WriteLine(IRIS.FunctionInt ("funcLong",routineName,7,8));

Console.Write("Calling funcDouble function. Result: ");
Console.WriteLine(IRIS.FunctionDouble ("funcDouble",routineName,7.56));

Console.Write("Calling funcProcedure class method. Result: ");
IRIS.Procedure("funcProcedure",routineName,67);
Console.WriteLine("(function funcProcedure returned successfully)");
```

# 3.3 Sample ObjectScript Methods and Functions

To run the examples, these ObjectScript class methods and functions must be compiled and available on the server:

### ObjectScript Class User.NativeTest

```
Class User.NativeTest Extends %Persistent
{
  ClassMethod cosBoolean(p1 As %Boolean) As %Boolean
  {
   Quit 0
  }
  ClassMethod cosBytes(p1 As %String) As %Binary
  {
   Quit $C(65,66,67,68,69,70,71,72,73,74)
  }
  ClassMethod cosString(p1 As %String) As %String
  {
   Quit "Hello "_p1
  }
  ClassMethod cosLong(p1 As %Integer, p2 As %Integer) As %Integer
  {
   Quit p1+p2
  }
  ClassMethod cosDouble(p1 As %Double) As %Double
  {
   Quit p1 * 100
  }
  ClassMethod cosVoid(p1 As %Integer)
  {
   Set ^p1=p1
   Quit
  }
}
```

### ObjectScript Routine NativeRoutine.mac

```
funcBoolean(p1) public {
    Quit 0
}
funcBytes(p1) public {
    Quit $C(65,66,67,68,69,70,71,72,73,74)
}
funcString(p1) public {
    Quit "Hello "_p1
}
funcLong(p1,p2) public {
    Quit p1+p2
}
funcDouble(p1) public {
    Quit p1 * 100
}
funcProcedure(p1) public {
    Set ^p1=p1
    Quit
}
```

# 4

# Native API for .NET Quick Reference

This is a quick reference for the InterSystems IRIS Native API for .NET, which consists of the following classes in InterSystems.Data.IRISClient.ADO:

- Class IRIS provides the main functionality.
- Class IRISIterator provides methods to navigate a global array.

**Note:** This chapter is intended as a convenience for readers of this book, but it is not the definitive reference for the Native API. For the most complete and up-to-date information, see the online class documentation under "Native API for .NET".

## 4.1 Class IRIS

For the most recent information on this class, see the IRIS section of "Native API for .NET" in the online documentation.

All methods listed in the following sections will throw an exception on encountering any kind of error. Throughout the API, the subscript argument `Object[] args` is defined as `params object[]`.

### 4.1.1 Constructor

Instances of class IRIS are created by calling IRIS.**CreateIRIS()**.

#### CreateIRIS()

Constructor IRIS.**CreateIRIS()** returns an instance of IRIS that uses the specified IRISConnection.

```
static IRIS CreateIRIS(IRISConnection conn)
```

*parameters:*

- `conn` — an instance of IRISConnection.

See "Introduction to Global Arrays" for more information and examples.

## 4.1.2 Method Details

### ClassMethodBool()

IRIS.**ClassMethodBool()** calls a class method, passing 0 or more arguments and returning an instance of bool?.

```
Nullable<bool> ClassMethodBool(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ClassMethodBytes()

IRIS.**ClassMethodBytes()** calls a class method, passing 0 or more arguments and returning an instance of byte[].

```
byte[] ClassMethodBytes(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ClassMethodDouble()

IRIS.**ClassMethodDouble()** calls a class method, passing 0 or more arguments and returning an instance of double?.

```
Nullable<double> ClassMethodDouble(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ClassMethodLong()

IRIS.**ClassMethodLong()** calls a class method, passing 0 or more arguments and returning an instance of long?.

```
Nullable<long> ClassMethodLong(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ClassMethodString()

IRIS.**ClassMethodString()** calls a class method, passing 0 or more arguments and returning an instance of string.

```
string ClassMethodString(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ClassMethodVoid()

IRIS.**ClassMethodVoid()** calls a class method with no return value, passing 0 or more arguments.

```
void ClassMethodVoid(string className, string methodName, Object[] args)
```

*parameters:*

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — 0 or more method arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### Close()

IRIS.**Close()** closes the IRIS object.

```
void Close()
```

See "Introduction to Global Arrays" for more information and examples.

### FunctionBool()

IRIS.**FunctionBool()** calls a function, passing 0 or more arguments and returning an instance of bool?.

```
Nullable<bool> FunctionBool(string functionName, string routineName, Object[] args)
```

*parameters:*

- `functionName` — name of the function to call
- `routineName` — name of the routine containing the function.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### FunctionBytes()

IRIS.**FunctionBytes()** calls a function, passing 0 or more arguments and returning an instance of byte[].

```
byte[] FunctionBytes(string functionName, string routineName, Object[] args)
```

*parameters:*

- `functionName` — name of the function to call
- `routineName` — name of the routine containing the function.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### FunctionDouble()

IRIS.**FunctionDouble()** calls a function, passing 0 or more arguments and returning an instance of double?.

```
Nullable<double> FunctionDouble(string functionName, string routineName, Object[] args)
```

*parameters:*

- `functionName` — name of the function to call
- `routineName` — name of the routine containing the function.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### FunctionInt()

IRIS.**FunctionInt()** calls a function, passing 0 or more arguments and returning an instance of long?.

```
Nullable<long> FunctionInt(string functionName, string routineName, Object[] args)
```

*parameters:*

- `functionName` — name of the function to call
- `routineName` — name of the routine containing the function.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### FunctionString()

IRIS.**FunctionString()** calls a function, passing 0 or more arguments and returning an instance of string.

```
string FunctionString(string functionName, string routineName, Object[] args)
```

*parameters:*

- `functionName` — name of the function to call
- `routineName` — name of the routine containing the function.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### GetBool()

IRIS.**GetBool()** gets the value of the global as a bool? (or `null` if node does not exist). Returns `false` if node value is empty string.

```
Nullable<bool> GetBool(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetBytes()

IRIS.**GetBytes()** gets the value of the global as a byte[] (or `null` if node does not exist).

```
byte[] GetBytes(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetDateTime()

IRIS.**GetDateTime()** gets the value of the global as a DateTime? (or `null` if node does not exist).

```
Nullable<DateTime> GetDateTime(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetDouble()

IRIS.**GetDouble()** gets the value of the global as a double? (or `null` if node does not exist). Returns `0.0` if node value is empty string.

```
Nullable<Double> GetDouble(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetInt16()

IRIS.**GetInt16()** gets the value of the global as an Int16? (or `null` if node does not exist). Returns 0 if node value is empty string.

```
Nullable<short> GetInt16(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetInt32()

IRIS.**GetInt32()** gets the value of the global as an Int32? (or `null` if node does not exist). Returns 0 if node value is empty string.

```
Nullable<int> GetInt32(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetInt64()

IRIS.**GetInt64()** gets the value of the global as an Int64? (or `null` if node does not exist). Returns 0 if node value is empty string.

```
Nullable<long> GetInt64(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetIRISIterator()

IRIS.**GetIRISIterator()** returns an IRISIterator object (see "Class IRISIterator") for the specified node with search direction set to `FORWARD`.

```
IRISIterator GetIRISIterator(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

See "Iterating Over a Set of Child Nodes" for more information and examples.

### GetIRISReverseIterator()

IRIS.**GetIRISReverseIterator()** returns an IRISIterator object (see "Class IRISIterator") for the specified node with search direction set to `BACKWARD`.

```
IRISIterator GetIRISReverseIterator(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

See "Iterating Over a Set of Child Nodes" for more information and examples.

### GetObject()

IRIS.**GetObject()** gets the value of the global as an Object (or `null` if node does not exist).

```
object GetObject(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetSingle()

IRIS.**GetSingle()** gets the value of the global as a Single? (Nullable<float>) or returns `null` if node does not exist. Returns `0.0` if node value is an empty string.

```
Nullable<float> GetSingle(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetString()

IRIS.**GetString()** gets the value of the global as a string (or `null` if node does not exist).

Empty string and null values require some translation. An empty string `""` in .NET is translated to the null string character `$CHAR(0)` in ObjectScript. A null in .NET is translated to the empty string in ObjectScript. This translation is consistent with the way .NET handles these values.

```
string GetString(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

### GetTLevel()

IRIS.**GetTLevel()** gets the level of the current nested transaction. Returns 1 if there is only a single transaction open. Returns 0 if there are no transactions open. This is equivalent to fetching the value of the **$TLEVEL** special variable.

```
Nullable<int> GetTLevel()
```

See "Transactions and Locking" for more information and examples.

### Increment()

IRIS.**Increment()** increments the specified global with the passed value. If there is no node at the specified address, a new node is created with `value` as the value. A null value is interpreted as 0. Returns the new value of the global node.

```
long Increment(long value, string globalName, Object[] subscripts)
```

*parameters:*

- `value` — long value to which to set this node (`null` value sets global to `0`).
- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

See "Creating, Updating, and Deleting Nodes" for more information and examples.

### IsDefined()

IRIS.**IsDefined()** returns a value indicating whether the specified node exists and if it contains a value.

```
int IsDefined(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts for this node

*return values:*

- `0` — the specified node does not exist
- `1` — the node exists and has a value
- `10` — the node is valueless but has subnodes
- `11` — the node has both a value and subnodes

See "Testing for Child Nodes and Node Values" for more information and examples.

### Kill()

IRIS.**Kill()** deletes the global node including any descendants.

```
void Kill(string globalName, Object[] subscripts)
```

*parameters:*

- `globalName` — global name
- `subscripts` — array of subscripts for this node

See "Creating, Updating, and Deleting Nodes" for more information and examples.

### Lock()

IRIS.**Lock()** locks the global, returns true on success. Note that this method performs an incremental Lock and not the implicit Unlock before Lock feature that is also offered in ObjectScript.

```
bool Lock(string lockMode, int timeout, string globalName, Object[] subscripts)
```

*parameters:*

- `lockMode` — Character `S` for shared Lock, `E` for escalating Lock, or `SE` for both. Default is empty string (exclusive and non-escalating)
- `timeout` — amount to wait to acquire the Lock in seconds
- `globalName` — global name
- `subscripts` — array of subscripts for this node

See "Transactions and Locking" for more information and examples.

### Procedure()

IRIS.**Procedure()** calls a procedure, passing 0 or more arguments.

```
void Procedure(string procedureName, string routineName, Object[] args)
```

*parameters:*

- `procedureName` — name of the procedure to call.
- `routineName` — name of the routine containing the procedure.
- `args` — 0 or more function arguments of supported types. Trailing arguments may be omitted.

See "Calling ObjectScript Methods and Functions" for more information and examples.

### ReleaseAllLocks()

IRIS.**ReleaseAllLocks()** releases all locks associated with the session.

```
void ReleaseAllLocks()
```

See "Transactions and Locking" for more information and examples.

### Set()

IRIS.**Set()** sets the current node to a value of a supported datatype (or `""` if the value is `null`). If there is no node at the specified node address, a new node will be created with the specified value. See "Creating, Updating, and Deleting Nodes" for more information.

```
void Set([type] value, string globalName, Object[] subscripts)
```

*parameters:*

- `value` — value of a supported datatype (`null` value sets global to `""`).
- `globalName` — global name
- `subscripts` — array of subscripts for this node

See "Creating, Updating, and Deleting Nodes" for more information and examples.

#### Notes on specific datatypes

The following datatypes have some extra features:

- string — empty string and null values require some translation. An empty string `""` in .NET is translated to the null string character `$CHAR(0)` in ObjectScript. A null in .NET is translated to the empty string in ObjectScript. This translation is consistent with the way .NET handles these values.

### TCommit()

IRIS.**TCommit()** commits the current transaction.

```
void TCommit()
```

See "Transactions and Locking" for more information and examples.

### TRollback()

IRIS.**TRollback()** rolls back all open transactions in the session.

```
void TRollback()
```

See "Transactions and Locking" for more information and examples.

### TRollbackOne()

IRIS.**TRollbackOne()** rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back.

```
void TRollbackOne()
```

See "Transactions and Locking" for more information and examples.

#### TStart()

IRIS.**TStart()** starts/opens a transaction.

```
void TStart()
```

See "Transactions and Locking" for more information and examples.

#### Unlock()

IRIS.**Unlock()** unlocks the global. This method performs an incremental Unlock, not the implicit Unlock-before-Lock feature that is also offered in ObjectScript. See "Transactions and Locking" for more information.

```
void Unlock(string lockMode, string globalName, Object[] subscripts)
```

*parameters:*

- `lockMode` — Character `S` for shared Lock, `E` for escalating Lock, or `SE` for both. Default is empty string (exclusive and non-escalating)
- `globalName` — global name
- `subscripts` — array of subscripts for this node

See "Transactions and Locking" for more information and examples.

# 4.2 Class IRISIterator

For the most recent information on this class, see the IRISIterator section of "Native API for .NET" in the online documentation.

All methods listed in the following sections will throw an exception on encountering any kind of error.

See "Finding Nodes in a Global Array" for more details and examples.

## 4.2.1 IRISIterator Constructor

Instances of IRISIterator are created by calling one of the following IRIS methods:

- IRIS.**GetIRISIterator()** — Returns an IRISIterator instance set to forward iteration.
- IRIS.**GetIRISReverseIterator()** — Returns an IRISIterator instance set to backward iteration.

See "Iterating Over a Set of Child Nodes" for more information and examples.

This class implements required IEnumerator method System.Collections.IEnumerator.**GetEnumerator()**, which is invoked when the iterator is used in a `foreach` loop.

## 4.2.2 Method and Property Details

#### Property CurrentSubscript

IRISIterator.**CurrentSubscript** gets the lowest level subscript for the node at the current iterator position. For example, if the iterator points to node *^myGlobal(23,"somenode")*, the returned value will be `"somenode"`.

```
object CurrentSubscript get;
```

See "Iteration in Conditional Loops" for more information and examples.

### Property Current

IRISIterator.**Current** gets the value of the node at the current iterator position. In a `foreach` loop, this value is also assigned to the current loop variable.

```
object Current get;
```

See "Iteration in Conditional Loops" for more information and examples.

### MoveNext()

IRISIterator.**MoveNext()** implements System.Collections.IEnumerator. It returns `true` if the next value was retrieved, `false` if there are no more values.

```
bool MoveNext()
```

See "Iteration in Conditional Loops" for more information and examples.

### Reset()

IRISIterator.**Reset()** can be called after completing a `foreach` loop to reset the iterator to its starting position, allowing it to be used again.

```
void Reset()
```

See "Iteration in Conditional Loops" for more information and examples.

### StartFrom()

IRISIterator.**StartFrom()** sets the iterator's starting position to the specified subscript. The subscript is an arbitrary starting point, and does not have to specify an existing node.

```
void  StartFrom(Object subscript)
```

See "Iterating Over a Set of Child Nodes" for more information and examples.