Using the Native API for Node.js

Version 2020.3
2021-02-04

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

Using the Native API for Node.js
InterSystems IRIS Data Platform Version 2020.3 2021-02-04

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 a lightweight *Native API for Node.js* for direct access to the native data structure of InterSystems databases. *Globals* are the 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 and data manipulation.
- 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.

Note: **Support for Node.js Relational Access**

InterSystems also supports Node.js relational access. The *node-odbc* open source Node.js module is supported on both Windows and UNIX® based systems. See "Support for Node.js Relational Access" in *Using the InterSystems ODBC Driver* for details.

More information about globals

Versions of the Native API are also available for Java, .NET, and Python:

- *Using the Native API for Java*
- *Using the Native API for .NET*
- *Using the Native API for Python*

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.

1

Introduction to the Native API

The *Native API for Node.js* 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 Node.js is implemented in the Iris class.

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 ObjectScript globals notation, the two nodes with values would be:

```
root('foo','SubFoo')
root('bar','lowbar','UnderBar')
```

The global name (`root`) is followed by a comma-delimited *subscript list* in parentheses. Together, they specify 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. A Connection object connects to the server and an instance of class Iris is created. Iris methods are used to create a global array, read the resulting persistent values from the database, and then delete the global array.

The NativeDemo Program

```
// Create a reference to the Native API module
  const IRISNative = require('intersystems-iris-native')

// Open a connection to the server and create an Iris object
  let connectionInfo = {
    host: '127.0.0.1',
    port: 51773,
    ns: 'USER',
    user: '_SYSTEM',
    pwd: 'SYS'
  };
  const conn = IRISNative.createConnection(connectionInfo);
  const native = conn.createIris();

// 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
  let subfoo = native.get('root', 'foo', 'SubFoo')
  let underbar = native.get('root', 'bar', 'lowbar', 'UnderBar')
  console.log('Created two values: ');
  console.log('   root("foo","SubFoo")=' + subfoo);
  console.log('   root("bar","lowbar","UnderBar")=' + underbar);
```

NativeDemo prints the following lines:

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

In this example, *IRISNative* is an instance of the Native API. Static method **createConnection()** defines a Connection object named *conn* that provides a connection to the database associated with the USER namespace. Native API methods perform the following actions:

- Connection.**createIRIS()** creates a new instance of Iris named *native*, which will access the database through server connection *conn*.
- Iris.**set()** creates new persistent nodes in database namespace USER.
- Iris.**get()** queries the database and returns the values of the specified nodes.
- Iris.**kill()** deletes the specified root 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 the previous section 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*.

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. 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 unsubscripted node at the base of the global array tree. The identifier for a root node is its global name with no subscripts.

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. (The global name plus the subscript list is the 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; a node that has child nodes can be valueless.

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. Valueless nodes are just pointers to lower level nodes.

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.

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.

Note: The examples in this chapter assume that an Iris object named *native* already exists and is connected to the server. The following code was used to create it:

```
const IRISNative = require('intersystems-iris-native')
let connectionInfo = {host:'127.0.0.1', port:51773, ns:'USER', user:'_SYSTEM', pwd:'SYS'};
const conn = IRISNative.createConnection(connectionInfo);
const native = conn.createIris();
```

For more information, see the Quick Reference entries for **createConnection()** and **createIris()**.

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.

Incrementing node values

Iris.**increment()** takes an integer *number* argument, increments the node value by that amount, and returns the incremented value. The initial target node value can be any supported numeric type, but the incremented value will be an integer. If there is no node at the target address, the method creates one and assigns the *number* argument as the value. This method uses a thread-safe atomic operation to change the value of the node, so the node is never locked.

In the following example, the first call to **increment()** creates new subnode *myGlobal('B')* with value `-2`, and assigns the returned value to *total*. The next two calls each increment by `-2` and assign the new value to *total*, and the loop exits when the node value is `-6`.

```
do {
  var total = native.increment(-2,'myGlobal', 'B');
} while (total > -6)
console.log('total = ' + total + ' and myGlobal('B') = ' + native.get('myGlobal', 'B'))

// Prints: total = -6 and myGlobal('B') = -6
```

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 node *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.

The entire global array could have been deleted immediately by calling **kill()** on the root node:

```
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 with next() — describes methods 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.**iterator()** returns an instance of Iterator for the child nodes of the specified target node.
- Iterator.**reversed()** — toggles iteration between forward and reverse collation order.
- Iterator.**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')
  let iterDogs = native.iterator('heros','dogs').reversed().startFrom('V');

  let output = '\nDog birth years: ';
  for ([key,value] of iterDogs) {
    output += key + ':' + value + '  ';
  };
  console.log(output);
```

This code prints the following output:

```
  Dog birth years: Lassie:1940  Hachiko:1923  Balto:1919
```

In this example, 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. 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 with next()

The Native API also supports the standard **next()** and return type iterator methods:

- Iterator.**next()** — positions the iterator at the next child node (if one exists) and returns an array containing properties `done` and `value`. The `done` property will be `false` if there are no more nodes. When an iterator is created, it defaults to the **entries()** return type.
- Iterator.**entries()** — sets return type to an array containing both the key (the top level subscript of the child node) and the node value. For example, the returned value for node *heros(,'dogs','Balto')* would be `['Balto',1919]`.
- Iterator.**keys()** — sets return type to return only the key (the top level subscript).
- Iterator.**values()** — sets return type to return only the node value.

In the following example, each call to the **next()** method sets variable *iter* to an array containing the current values for the iterator `done` and `value` properties. Since the **keys()** method was called when the iterator was created, the `value` property will contain only the key (top level subscript) for the current child node of *heros('dogs')*.

Use next() to list the subscripts under node heros('dogs')

```
// Get a list of child subscripts under node heros('dogs')
  let iterDogs = native.iterator('heros','dogs').keys();
  let output = "\nSubscripts under node heros('dogs'): ";

  let iter = iterDogs.next();
  while (!iter.done) {
    output += iter.value + ' ';
    iter = iterDogs.next();
  }
  console.log(output);
```

This code prints the following output:

```
Subscripts under node heros('dogs'): Balto Hachiko Lassie Whitefang
```

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 set of child nodes. 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:

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

The following example consists of two methods:

- **findAllHeros()** iterates over child nodes of the current node, and calls **testNode()** for each node. Whenever **testNode()** indicates that the current node has child nodes, **findAllHeros()** creates a new iterator for the next level of child nodes.
- **testNode()** 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 prints node information for each node.

Method findAllHeros()

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.

```
function findAllHeros() {
  const root = 'heros';
  console.log('List all subnodes of root node '+root+':\n'+root)
  let iterRoot = native.iterator(root);
  let hasChild = false;

  // Iterate over children of root node heros
  for ([sub1,value] of iterRoot) {
    hasChild = testNode(value,root,sub1);

    // Iterate over children of heros(sub1)
    if (hasChild) {
      let iterOne = native.iterator(root,sub1);
      for ([sub2,value] of iterOne) {
        hasChild = testNode(value,root,sub1,sub2);

        // Iterate over children of heros(sub1,sub2)
        if (hasChild) {
          let iterTwo = native.iterator(root,sub1,sub2);
          for ([sub3,value] of iterTwo) {
            testNode(value,root,sub1,sub2,sub3); //no child nodes below level 3
          }
        } //end level 2
      }
    } //end level 1
  } // end main loop
} // end findAllHeros()
```

Method testNode()

```
function testNode(value, root, ...subs) {

// Test for values and child nodes
  let state = native.isDefined(root,...subs);
  let hasValue = (state%10 > 0); // has value if state is 1 or 11
  let hasChild = (state > 9);    // has child if state is 10 or 11

// format the node address output string
  let subList = Array.from(subs);
```

```
    let level = subList.length-1;
    let indent = '   ' + String('      ').slice(0,(level*2));
    let address = indent + root+'(' + subList.join() + ')';

  // Add node value to string and note special cases
    if (hasValue) { // ignore valueless nodes
      address += ' = ' + value;
      for (name of ['Timmy','Dolly']) {
        if (name == subList[level]) {
          address += ' (not a dog!)'
        }
      }
    }
    console.log(address);
    return hasChild;
  }
}
```

This method will write the following lines:

```
List all subnodes of root node heros:
heros
  heros(dogs)
    heros(dogs,Balto) = 1919
    heros(dogs,Hachiko) = 1923
    heros(dogs,Lassie) = 1940
      heros(dogs,Lassie,Timmy) = 1954  (not a dog!)
    heros(dogs,Whitefang) = 1906
  heros(sheep)
    heros(sheep,Dolly) = 1996  (not a dog!)
```

The output of **testNodes()** includes some nodes that were not found in previous examples because they are not child nodes of *heros('dogs')*:

- *heros('dogs','Lassie','Timmy')* is a child of `Lassie`, not `dogs`.
- *heros('sheep','Dolly')* is a child of `sheep`, not `dogs`.

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.**getotalevel()** — 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**

```
  const node = 'myGlobal';
  console.log('Set three values in three different transaction levels:');
  for (let i=1; i<4; i++) {
    native.tStart();
    let lvl = native.getTLevel()
    native.set(('Value'+lvl), node, lvl);
    let val = '<valueless>'
    if (native.isDefined(node,lvl)%10 > 0) val = native.get(node,lvl);
    console.log('  ' + node + '(' + i + ') = ' + val + ' (tLevel is '+ lvl + ')');
  }
// Prints: Set three values in three different transaction levels:
//         myGlobal(1) = Value1 (tLevel is 1)
//         myGlobal(2) = Value2 (tLevel is 2)
//         myGlobal(3) = Value3 (tLevel is 3)

  console.log('Roll back two levels and commit the level 1 transaction:');
  let act = ['  tRollbackOne','  tRollbackOne','       tCommit'];
  for (let i=3; i>0; i--) {
    if (i>1) {native.tRollbackOne();} else {native.tCommit();}
    let val = '<valueless>'
    if (native.isDefined(node,i)%10 > 0) val = native.getString(node,i);
    console.log(act[3-i]+' (tLevel='+native.getTLevel()+'): '+node+'('+i+') = '+val);
  }

// Prints: Roll back two levels and commit the level 1 transaction:
//           tRollbackOne (tLevel=2): myGlobal(3) = <valueless>
//           tRollbackOne (tLevel=1): myGlobal(2) = <valueless>
//                tCommit (tLevel=0): myGlobal(1) = Value1
```

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:

- 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 **alert()** 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('exclusive node','nodeOne');
  native.set('shared node','nodeTwo');

// unlike global names, lock references *must* start with circumflex
  const nodeOneRef = '^nodeOne';
  const nodeTwoRef = '^nodeTwo';

  try {
    native.tStart();
    native.lock('E',10,nodeOneRef,''); // lock nodeOne exclusively
    native.lock('S',10,nodeTwoRef,''); // lock nodeTwo shared
    console.log('Exclusive lock on nodeOne and shared lock on nodeTwo');

    alert('Press return to release locks individually');
    native.unlock('D',nodeOneRef,''); // release nodeOne after transaction
    native.unlock('I',nodeTwoRef,''); // release nodeTwo immediately

    alert('Press return to commit transaction');
    native.tCommit();
  }
  catch { console.log('error'); }
```

Using non-incremental locking in transactions

```
// lock nodeOne non-incremental, nodeTwo shared non-incremental
  native.lock('',10,nodeOneRef,'');

  alert('Exclusive lock on nodeOne, return to lock nodeOne non-incrementally');
  native.lock('S',10,nodeTwoRef,'');

  alert('Verify that only nodeTwo is now locked, then press return');
```

Using releaseAllLocks() in transactions to release all incremental locks

```
// lock nodeOne shared incremental, nodeTwo exclusive incremental
  native.lock('SE',10,nodeOneRef,'');
  native.lock('E',10,nodeTwoRef,'');

  alert('Two locks are held (one with lock count 2), return to release both locks');
  native.releaseAllLocks();

  alert('Verify both locks have been released, then press return');
```

3

Calling ObjectScript Methods and Functions

This chapter describes methods of class Iris that allow an application to call class methods and functions from user defined ObjectScript classes and routines.

- 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 called 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 array containing `0` or more method arguments:

- Iris.**ClassMethodValue()** — calls a classmethod and gets the return value.
- Iris.**ClassMethodVoid()** — calls a classmethod and discards any returned value.

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 several datatypes from an ObjectScript test class named User.NativeTest. (For a listing, see "Sample ObjectScript Methods and Functions" at the end of this chapter).

Calling class methods from ObjectScript class User.NativeTest

In this example, assume that *native* is an existing instance of class Iris, and is currently connected to the server.

```
const className = 'User.NativeTest';
let cmValue = "";

cmValue = native.classMethodValue(className,'cmBoolean',false);
console.log(className+'.cmBoolean() returned value: ' + cmValue);

cmValue = native.classMethodValue(className,'cmBytes','byteArray');
console.log(className+'.cmBytes() returned value: ' + cmValue);

cmValue = native.classMethodValue(className,'cmString','Test String');
console.log(className+'.cmString() returned value: ' + cmValue);
```

```
cmValue = native.classMethodValue(className,'cmLong',7,8);
console.log(className+'.cmLong() returned value: ' + cmValue);

cmValue = native.classMethodValue(className,'cmDouble',7.56);
console.log(className+'.cmDouble() returned value: ' + cmValue);

try {
  native.classMethodVoid(className,'cmVoid',67);
  cmValue = native.get("p1");
}
catch {cmValue = 'method failed.'}
console.log(className+'.cmVoid() set global array p1 to value: ' + cmValue);
```

3.2 Function Calls

The following Native API methods call user-defined ObjectScript functions or procedures (see "Callable User-defined Code Modules" in *Using ObjectScript*). They take string arguments for *className* and *methodName*, plus an array containing `0` or more method arguments:

- Iris.**function()** — calls a user-defined function and gets the return value.
- Iris.**procedure()** calls a user-defined procedure and discards any returned value.

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 several datatypes from an ObjectScript test routine named **NativeRoutine**. (For a listing of NativeRoutine.mac, see "Sample ObjectScript Methods and Functions" at the end of this chapter).

Calling functions from NativeRoutine.mac

In this example, assume that *native* is an existing instance of Iris, and is currently connected to the server.

```
const routineName = 'NativeRoutine';
let fnValue = "";

fnValue = native.function('funcBoolean',routineName,false);
console.log(routineName+'.funcBoolean() returned value: ' + fnValue);

fnValue = native.function('funcBytes',routineName,'byteArray');
console.log(routineName+'.funcBytes() returned value: ' + fnValue);

fnValue = native.function("funcString",routineName,"Test String");
console.log(routineName+'.funcString() returned value: ' + fnValue);

fnValue = native.function('funcLong',routineName,7,8);
console.log(routineName+'.funcLong() returned value: ' + fnValue);

fnValue = native.function('funcDouble',routineName,7.56);
console.log(routineName+'.funcDouble() returned value: ' + fnValue);

try {
  native.procedure("funcProcedure",routineName,67);
  fnValue = native.get("p1");
}
catch {fnValue = 'procedure failed.'}
console.log(routineName+'.funcVoid() set global array p1 to value: ' + fnValue);
```

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 cmBoolean(p1 As %Boolean) As %Boolean
  {
   Quit 0
  }
  ClassMethod cmBytes(p1 As %String) As %Binary
  {
   Quit $C(65,66,67,68,69,70,71,72,73,74)
  }
  ClassMethod cmString(p1 As %String) As %String
  {
   Quit "Hello "_p1
  }
  ClassMethod cmLong(p1 As %Integer, p2 As %Integer) As %Integer
  {
   Quit p1+p2
  }
  ClassMethod cmDouble(p1 As %Double) As %Double
  {
   Quit p1 * 100
  }
  ClassMethod cmVoid(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 Quick Reference for Node.js

This chapter is a quick reference for the following classes, which are all members of module external:"intersystems-iris-native"

- Class Connection provides a connection to the server.
- Class Iris provides the main functionality.
- Class Iterator 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 on these classes, see the online API documentation.

4.1 List of Methods by Usage

Class Connection

The Connection class encapsulates a connection to the server. Instances of Connection are created and connected to the server by external:"intersystems-iris-native" method **createConnection()**.

- **close()** — closes the connection.
- **createIris()** — creates an instance of Iris for this connection.
- **isClosed()** — returns `true` if the connection is closed.
- **isUsingSharedMemory()** — returns `true` if the connection is using shared memory.

Class Iris

The Iris class provides most of the Native API functionality. Instances of Iris are created by Connection.**createIris()**. Methods are listed below, organized by usage:

Class Iris: Global Iteration and Management

- **getAPIVersion()** — returns the version string for this version of the Native API.
- **getServerVersion()** — returns the version string for the currently connected server.
- **increment()** — increments the value of a global node by the specified amount.
- **isDefined()** — checks if a global node exists, and if it contains data.
- **iterator()** — returns an instance of Iterator.

- **kill()** — kills the global node including any descendants.

Class Iris: Node Value Accessors

- **get()** — returns the specified node value
- **getBoolean()** — returns a node value as boolean
- **getBytes()** — returns a node value as an ArrayBuffer
- **getNumber()** — returns a node value as a number
- **getString()** — returns a node value as a string

Class Iris: Transactions and Locking

- **lock()** — performs an incremental lock on the global, returns `true` on success.
- **unlock()** — performs an immediate or deferred unlock on the global.
- **releaseAllLocks()** — releases all locks associated with the session.
- **getTLevel()** — returns the current transaction level (`0` if not in a transaction).
- **tCommit()** — commits the current transaction and decrements the transaction level.
- **tRollback()** — rolls back all open transactions in the session.
- **tRollbackOne()** — rolls back the current level transaction only.
- **tStart()** — starts a transaction and increments the transaction level.

Class Iris: ClassMethod and Function Calls

- **classMethodValue()** — calls a user defined ObjectScript method and gets the returned value
- **classMethodVoid()** — calls a user defined ObjectScript method, ignoring any returned value
- **function()** — calls a function of a user defined ObjectScript routine and gets the returned value
- **procedure()** — calls a procedure of a user defined ObjectScript routine

Class Iterator

The Iterator class provides methods to iterate over a set of nodes. Instances of Iterator are created by Iris.**iterator()**.

- **next()** — positions the iterator at the next sibling node.
- **startFrom()** — sets starting position to the specified subscript.
- **reversed()** — toggles iteration between forward and reverse collation order.
- **entries()** — sets return type to an array containing subscript and node value.
- **keys()** — sets return type to return the node subscript (key) only.
- **values()** — sets return type to return the node value only.

4.2 Class Connection

Class Connection is a member of external:"intersystems-iris-native". Instances of Connection are created by intersystems-iris-native method **createConnection()**.

createConnection()

intersystems-iris-native **createConnection()** establishes a connection to the server.

```
(static) createConnection(connectionInfo)   {external:"intersystems-iris-native".Connection}
```

parameters:

- `connectionInfo` — Object containing connection properties. Valid properties of *connectionInfo* are:
  - `host` — *required* string containing the address of the host machine
  - `port` — *required* integer specifying the port number
  - `ns` — *required* string containing the database namespace
  - `user` — string containing the user name
  - `pwd` — string containing the password
  - `sharedmemory` — boolean indicating whether to use shared memory if available (default is `true`).
  - `timeout` — integer specifying the number of milliseconds to wait before the connection attempt times out (default is `10000`).
  - `logfile` — string specifying the full path and name of a log file for this connection. If not specified, no log file will be written.

  For example, the following code fragment defines all properties of *connectionInfo* and creates an instance of Connection named `conn`:

```
const IRISNative = require('intersystems-iris-native')

let connectionInfo = {
  host: '127.0.0.1',
  port: 51773,
  ns: 'USER',
  user: '_SYSTEM',
  pwd: 'SYS',
  sharedmemory: true
  timeout: 5,
  logfile: 'C:\temp\mylogfile.log'
};

const conn = IRISNative.createConnection(connectionInfo);
```

close()

Connection.**close()** closes the connection to the server.

```
close()
```

createIris()

Connection.**createIris()** returns an instance of class Iris connected to the server by this Connection.

```
createIris()    {external:"intersystems-iris-native".Iris}
```

isClosed()

Connection.**isClosed()** returns `true` if the connection is currently closed, `false` if the connection is currently open.

```
isClosed()   {boolean}
```

isUsingSharedMemory()

Connection.**isUsingSharedMemory()** returns `true` if the connection is using shared memory, `false` if it is not.

```
isUsingSharedMemory()   {boolean}
```

4.3 Class Iris

Class Iris is a member of external:"intersystems-iris-native". Instances of Iris are created by calling Connection.**createIris()**.

The main IRIS Native API is implemented by the IRIS class. Functions implemented by IRIS are global functions, function and procedure calls, locks, and transactions.

classMethodValue()

Iris.**classMethodValue()** calls a class method, passing 0 or more arguments and returning the called method's return value..

```
classMethodValue(methodName, ...args)   {any}
```

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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

classMethodVoid()

Iris.**classMethodVoid()** calls a class method, passing 0 or more arguments and returning nothing.

```
classMethodVoid(methodName, ...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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

function()

Iris.**function()** calls a function, passing 0 or more arguments and returning the called function's return value.

```
function(functionName, routineName, ...args)   {any}
```

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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

get()

Iris.**get()** returns the value of the global node. Returns `false` if node value is an empty string, or `null` if the specified node address does not have a value.

```
get(globalName, ...subscripts)   {any}
```

parameters:

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

getAPIVersion()

Iris.**getAPIVersion()** returns the Native API version string.

```
getAPIVersion()   {string}
```

getBoolean()

Iris.**getBoolean()** returns the node value as a boolean. Returns `false` if node value is an empty string, or `null` if the specified node address does not have a value.

```
getBoolean(globalName, ...subscripts)   {boolean}
```

parameters:

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

getBytes()

Iris.**getBytes()** returns the node value as an ArrayBuffer. Returns `false` if node value is an empty string, or `null` if the specified node address does not have a value.

```
getBytes(globalName, ...subscripts)   {ArrayBuffer}
```

parameters:

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

getNumber()

Iris.**getNumber()** returns the node value as a number. Returns `false` if node value is an empty string, or `null` if the specified node address does not have a value.

```
getNumber(globalName, ...subscripts)   {number}
```

parameters:

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

getServerVersion()

Iris.**getServerVersion()** returns the version string of the currently connected server.

```
getServerVersion()   {string}
```

getString()

Iris.**getString()** returns the node value as a string. Returns `false` if node value is an empty string, or `null` if the specified node address does not have a value.

```
getString(globalName, subscripts)   {string}
```

parameters:

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

getTLevel()

Iris.**getTLevel()** returns 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. See "Transactions and Locking" for more information and examples.

```
getTLevel()   {number}
```

increment()

Iris.**increment()** increments the specified node by the integer value of `incrementBy`. If the specified node address does not have a value, then the node will be created and its value set to the value of `incrementBy`. Returns the new integer value of the node.

```
increment(incrementBy, globalName, ...subscript)   {number}
```

parameters:

- `incrementBy` — numeric value to which to set this node (`null` value sets global to `0`, decimal value will be truncated to an integer).
- `globalName` — global name
- `subscripts` — array of subscripts specifying the target node

isDefined()

Iris.**isDefined()** checks if a global exists and contains data (see **$DATA**). Returns 0 if the node does not exist, 1 if the global node exists and contains data. 10 if the node is valueless but has descendants. 11 if it has data and descendants. See "Testing for Child Nodes and Node Values" for more information and examples.

```
isDefined(globalName, ...subscript)   {number}
```

parameters:

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

iterator()

Iris.**iterator()** returns an Iterator object (see "Class Iterator") for the specified node.

```
iterator(globalName, ...subscript)   {external:"intersystems-iris-native".Iterator}
```

parameters:

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

kill()

Iris.**kill()** kills the global node including any descendants.

```
kill(globalName, ...subscript)
```

parameters:

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

lock()

Iris.**lock()** locks the global, returns `true` on success. This method performs an incremental lock, not the implicit unlock-before-lock feature that is offered in ObjectScript.

```
lock(lockType, timeout, lockReference, ...subscript)   {boolean}
```

parameters:

- `lockMode` — character `S` for shared lock or `E` for escalating lock.
- `timeout` — amount to wait to acquire the lock in seconds
- `lockReference` — a string starting with a circumflex (`^`) followed by the global name (for example, `^myGlobal`, *not* just `myGlobal`).

  NOTE: Unlike the `globalName` parameter used by most methods, the `lockReference` parameter *must* be prefixed by a circumflex. Only **lock()** and **unlock()** use `lockReference` instead of `globalName`.
- `subscripts` — array of subscripts for this node

procedure()

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

```
procedure(procedureName, routineName, ...args)
```

parameters:

- `procedureName` — name of the procedure to call.
- `routineName` — name of the routine containing the procedure.
- `args` — 0 or more procedure arguments of supported types (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

releaseAllLocks()

Iris.**releaseAllLocks()** releases all locks associated with the session.

```
releaseAllLocks()
```

set()

Iris.**set()** sets the current node to a value of a supported datatype (or `""` if the value is `null`):

```
set(value, globalName, ...subscript)
```

parameters:

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

tCommit()

Iris.**tCommit()** commits the current transaction and decrements the transaction level. See "Transactions and Locking" for more information and examples.

```
tCommit()
```

tRollback()

Iris.**tRollback()** rolls back all open transactions in the session. See "Transactions and Locking" for more information and examples.

```
tRollback()
```

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

```
tRollbackOne()
```

tStart()

Iris.**tStart()** starts a transaction and increments the transaction level. See "Transactions and Locking" for more information and examples.

```
tStart()
```

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.

```
unlock(lockMode, lockReference, ...subscript)
```

parameters:

- `lockMode` — string containing lock type (`S` for Shared or `E` for Escalating) and unlock type (`I` for an Immediate unlock or `D` for Deferred).
- `lockReference` — a string starting with a circumflex (`^`) followed by the global name (for example, `^myGlobal`, *not* just `myGlobal`).

  NOTE: Unlike the `globalName` parameter used by most methods, the `lockReference` parameter *must* be prefixed by a circumflex. Only **lock()** and **unlock()** use `lockReference` instead of `globalName`.
- `subscripts` — array of subscripts for this node

4.4 Class Iterator

Class Iterator is a member of external:"intersystems-iris-native". Instances of Iterator are created by calling Iris.**iterator()**. See "Finding Nodes in a Global Array" for more details and examples.

next()

Iterator.**next()** positions the iterator at the next sibling node in collation order and returns an object containing *done* and *value* properties. If the iterator is at end then *done* is `true`, otherwise it is `false`.

```
next()   {any}
```

When *done* is `false`, the return type of the *value* property can be set by any of the following methods:

- **entries()** causes *value* to return as an array where *value(0)* is the subscript and *value(1)* is the node value (this array is the default when the iterator is created).
- **keys()** causes *value* to return only the current subscript.
- **values()** causes *value* to return only the value of the current node.

startFrom()

Iterator.**startFrom()** sets the iterator's starting position to the specified subscript. The starting position does not have to be a valid subnode. Returns `this` for chaining.

```
startFrom(subscript)   {external:"intersystems-iris-native".Iterator}
```

parameter:

- `subscript` — the subscript to use as a starting point. Does not have to specify an existing node.

The iterator will not point to a node until you call **next()** to advance to the next existing node in collating order.

reversed()

Iterator.**reversed()** toggles iteration direction between forward and reverse collation order. By default, the iterator is set to forward iteration when it is defined. Returns `this` for chaining.

```
reversed()   {external:"intersystems-iris-native".Iterator}
```

entries()

Iterator.**entries()** specifies that **next()**.*value* should be an array containing the node subscript (*value(0)*) and node value (*value(1)*). Returns `this` for chaining.

```
entries()   {external:"intersystems-iris-native".Iterator}
```

keys()

Iterator.**keys()** specifies that **next()**.*value* should contain only the node subscript (key). Returns `this` for chaining.

```
keys()   {external:"intersystems-iris-native".Iterator}
```

values()

Iterator.**values()** specifies that **next()**.*value* should contain only the node value. Returns `this` for chaining.

```
values()   {external:"intersystems-iris-native".Iterator}
```