Using the Native API for Java

Version 2018.1
2019-02-14

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

Using the Native API for Java
InterSystems IRIS Data Platform Version 2018.1 2019-02-14

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 lightweight Java APIs for easy database access to relational tables, objects, and multidimensional storage. See *Using Java with the InterSystems JDBC Driver* for relational table access and *Persisting Java Objects with InterSystems XEP* for object access. This book describes the Native API for multidimensional storage.

The *Native API* provides direct access to *globals* (tree-based sparse arrays used to implement the multidimensional storage model). These native data structures provide very fast, flexible storage and retrieval. InterSystems IRIS uses globals to structure data 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 jdbc.IRIS class (which implements the Native API), 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 class methods and functions from the InterSystems IRIS class library.
- Quick Reference for the Native API — provides a brief description of each method in the Native API.

There is also a detailed Table of Contents.

Related Documents

The following documents contain related material:

- *Using Java with the InterSystems JDBC Driver* — describes how use the InterSystems JDBC driver, which provides SQL access to external data sources.
- *Persisting Java Objects with InterSystems XEP* — describes how to store and retrieve Java objects using the InterSystems IRIS event persistence API (XEP).
- *Using Globals* — describes how InterSystems IRIS stores its data in multidimensional sparse arrays, known as globals

1

Introduction to the Native API

The *Native API* is a lightweight Java 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 is implemented in the com.intersystems.jdbc.IRIS class as an extension to the InterSystems JDBC driver (see "InterSystems Java Connectivity Options" in *Using Java JDBC with InterSystems IRIS* for details).

This chapter discusses the following topics:

- Introduction to Global Arrays — introduces global array concepts and terminology.
- The NativeDemo Program — provides a simple demonstration of how the Native API is used.

1.1 Introduction to Global Arrays

An InterSystems IRIS™ global array, like all sparse arrays, is a tree structure rather than a sequentially numbered list. The basic concept behind sparse arrays can be illustrated by analogy to the Java package naming convention. Each class in a package is uniquely identified by a namespace made up of a series of identifiers. For example, consider a package containing only two classes, main.foo.SubFoo and main.bar.UnderBar. The five identifiers that make up these namespaces could be viewed as elements of a sparse array:

```
main -->|--> foo --> SubFoo
        |--> bar --> UnderBar
```

Classes could be created that use the other possible namespaces (for example, main or main.bar), but no resources are wasted if those classes do not exist.

Like Java classes, the elements of a global array are uniquely identified by a namespace consisting of an arbitrary number of identifiers. All namespaces in the global array structure are referred to as *nodes*. 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*.

The root identifier (like main in the Java package) is referred to as the *global name*. All other identifiers in the namespace are called *subscripts*. The complete namespace of the node (global name plus subscripts) is the *node address*.

In ObjectScript notation, a node address is symbolized by a circumflex (^) followed by the global name and a comma-delimited *subscript list* in parentheses. For example, given a global array using the same identifiers as the Java package example, the nodes in the array would be represented as follows:

```
(valueless root node)   ^main
(valueless node)           ^main("foo")
(node with value)             ^main("foo","SubFoo") = <value>
(valueless node)           ^main("bar")
(node with value)             ^main("bar","UnderBar") = <value>
```

The *root node* of the array is simply *^main*, with no subscript list. The two nodes containing data are *^main("foo","SubFoo")* and *^main("bar","UnderBar")*. Nodes *^main*, *^main("foo")* and *^main("bar")* are all valueless.

All descendants of a given node are referred to as *subnodes* of that node. For example, *^main("foo","SubFoo")* is a subnode of *^main("foo")*, and all four subscripted nodes are subnodes of *^main*.

The term *node level* refers to the number of subscripts in the subscript list. For example, *^main("bar","UnderBar")* is a level 2 node. The address has a level 1 subscript (`"bar"`), and a level 2 subscript (`"UnderBar"`).

For more detailed information on the structure of global arrays, see "Logical Structure of Globals" in *Using Globals*.

1.2 The NativeDemo Program

This section describes a Native API application that creates and accesses a global array. In NativeDemo, a new jdbc.IRISConnection object is created and connected to the USER namespace on the InterSystems IRIS server. The connection object is used to create an instance of the jdbc.IRIS class. The program adds a new global array with two nodes to the database, reads the new persistent values from the database, and then deletes them before closing the connection and terminating.

The NativeDemo Program

```
package nativedemo;
import com.intersystems.jdbc.*;

public class NativeDemo {
  public static void main(String[] args) throws Exception {
    String connStr = "jdbc:IRIS://127.0.0.1:51773/USER";
    String user = "_SYSTEM";
    String password = "SYS";
    try {
      System.out.println("\nCreating the database connection... ");
      IRISConnection conn = (IRISConnection)
java.sql.DriverManager.getConnection(connStr,user,password);
      System.out.println("Creating the IRIS instance... ");
      IRIS dbnative = IRIS.createIRIS(conn);

      dbnative.set("Hello world","globalArray","sub1");
      System.out.println("Created subnode ^globalArray(\"sub1\") with value \""
                         + dbnative.getString("globalArray","sub1") + "\"");
      dbnative.set("value2","globalArray");
      System.out.println("Assigned value \"" + dbnative.getString("globalArray") + "\" to root
 node.");

      dbnative.kill(globalName);   // delete entire global array ^globalArray
      dbnative.close();
      conn.close();
    }
    catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }// end main()
} // end class NativeDemo
```

NativeDemo prints the following lines:

```
Creating the database connection...
Creating the IRIS instance...
Created subnode ^globalArray("sub1") with value "Hello world"
Assigned value "value2" to root node.
```

In this example, JDBC and Native API methods perform the following actions:

- java.sql.DriverManager.**getConnection()** creates a new Connection, and the returned connection is cast to an IRISConnection object named *conn*. (jdbc.IRISConnection is the InterSystems implementation of the standard JDBC Connection class). The *conn* object provides a JDBC connection to the database associated with the USER namespace.
- IRIS.**createIRIS()** creates a new instance of jdbc.IRIS named *dbnative*, which will access the database through *conn*.
- IRIS.**set()** creates a new persistent node in the database. The call specifies value `"Hello world"`, global name `"globalArray"`, and subscript `"sub1"` creating subnode *^globalArray("sub1")*. In the print statement, a call to IRIS.**getString()** queries the database and returns the value of the subnode.
- Root node *^globalArray* (with no subscripts) now exists because it contains a subnode, even though it does not contain a value. The second call to **set()** assigns string `"value2"` to *^globalArray*. In the print statement, **getString()** returns the value of *^globalArray* from the database.
- IRIS.**kill()** deletes the specified node and all of its subnodes from the database. Since both nodes of the global array *^globalArray* would persist in the database after the connection is closed, we delete the entire array at once by calling to **kill()** on root node *^globalArray*.
- This little program assumes that it has exclusive use of the system, so **close()** methods for both *dbnative* and *conn* are called. A real world application might be using several different IRISConnection objects, all of which would be closed because they all use the same underlying connection.

All of these methods are discussed in detail later in this book. See the next chapter (“Working with Global Arrays”) for details about specifying the subscripts of a node reference, creating a node, and changing a node value.

2

Working with Global Arrays

This chapter covers the following topics:

- Creating and Deleting Nodes — demonstrates how to create, change, or delete nodes in a global array.
- Finding Nodes in a Global Array — describes the 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 and Deleting Nodes

The Native API (class jdbc.IRIS) contains numerous methods to connect to a database and read data from the global arrays stored there, but it contains only three methods that can actually make changes in the database: **set()**, **increment()**, and **kill()**. This section describes how these methods are used.

Creating and Changing Nodes with set() and increment()

The following jdbc.IRIS methods can be used to create a persistent node with a specified value, or to change the value of an existing node:

- **set()** — takes a *value* argument and stores the value at the target address. If no node exists at that address, a new one is created when the value is stored. The *value* argument can be Boolean, byte[], Double, Integer, Long, Short, String, Date, Time, Timestamp, plus Object and abstract classes InputStream, and Reader.
- **increment()** — takes an Integer *number* argument, increments the target node value by that amount, and returns the incremented value as a Long. Unlike **set()**, it uses a thread-safe atomic operation to change the value of the node, so the node is never locked. The target node value can be Double, Integer, Long, or Short. If there is no node at the target address, the method creates one and assigns the *number* argument as the value.

The following example uses both **set()** and **increment()** to create and change node values:

Creating and Deleting Nodes: Setting and incrementing node values

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`.

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

The **increment()** method can both create and increment a node with a numeric value. Unlike **set()**, it uses a thread-safe atomic operation that never locks the node.

In the following example, **increment()** is called three times. The first call creates new subnode *^myGlobal("B")* with value `-2`. The next two calls each change the existing value by `-2`, resulting in a final value of `-6`:

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

Deleting Nodes with kill()

A node is deleted from the database when it is valueless and has no subnodes with values. When the last value is deleted from a global array, the entire global array is deleted.

- **kill()** — deletes the specified node and all of its subnodes. If the root node is specified, the entire global array is deleted. This method is equivalent to the InterSystems IRIS inclusive KILL command.

The following example assumes that the global array initially contains the following nodes:

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

We could delete the entire global array with one command by specifying only the root node:

```
dbnative.kill("myGlobal");
```

The following example also deletes the entire array in a different way.

Creating and Deleting Nodes: Using kill() to delete a group of nodes or an entire global array

In this example, global array *^myGlobal* will be deleted because two separate calls to **kill()** delete all subnodes with values:

```
  dbnative.kill("myGlobal","A",1);

// Now only these nodes are left:
//   ^myGlobal = <valueless node>
//     ^myGlobal("B") = 0
//       ^myGlobal("B",1) = 0
//       ^myGlobal("B",2) = 0

  dbnative.kill("myGlobal","B");

// Array no longer exists because all values have been deleted.
```

The first call to **kill()** deletes subnode *^myGlobal("A",1)*. Node *^myGlobal("A")* no longer exists because it is valueless and now has no subnodes.

The second call to **kill()** deletes node *^myGlobal("B")* and both of its subnodes. Since root node *^myGlobal* is valueless and now has no subnodes, the entire global array is deleted from the database.

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.

- Finding Subnodes on All Levels — describes how to test for the existence of subnodes and iterate over all subnodes regardless of node level.

2.2.1 Iterating Over a Set of Child Nodes

Child nodes are sets of subnodes immediately under the same parent node. Any child of the current target node can be addressed by adding only one subscript to the target address. All child nodes under the same parent are *sibling nodes* of each other. For example, the following global array has six sibling nodes under parent node *^myNames("people")*:

```
^myNames                                (valueless root node)
   ^myNames("people")                   (valueless level 1 node)
      ^myNames("people","Anna") = 2     (first level 2 child node)
      ^myNames("people","Julia") = 4
      ^myNames("people","Misha") = 5
      ^myNames("people","Ruri") = 3
      ^myNames("people","Vlad") = 1
      ^myNames("people","Zorro") = -1   (this node will be deleted in example)
```

Note: **Collation Order**

The iterator returns nodes in *collation order* (alphabetical order in this case: `Anna`, `Julia`, `Misha`, `Ruri`, `Vlad`, `Zorro`). This is not a function of the iterator. When a node is created, InterSystems IRIS automatically stores it in the collation order specified by the storage definition. The nodes in this example would be stored in the order shown, regardless of the order in which they were created.

This section demonstrates the following methods:

- *Methods used to create an iterator and traverse a set of child nodes*
  - jdbc.IRIS.**getIRISIterator()** returns an instance of IRISIterator for the global starting at the specified node.
  - IRISIterator.**next()** returns the subscript for the next sibling node in collation order.
  - IRISIterator.**hasNext()** returns `true` if there is another sibling node in collation order.

- *Methods that act on the current node*
  - IRISIterator.**getValue()** returns the current node value.
  - IRISIterator.**getSubscriptValue()** returns the current subscript (same value as the last successful call to **next()**).
  - IRISIterator.**remove()** deletes the current node and all of its subnodes.

The following example iterates over each child node under *^myNames("people")*. It prints the subscript and node value if the value is `0` or more, or deletes the node if the value is negative:

Finding all sibling nodes under *^myNames("people")*

```
// Read child nodes in collation order while iter.hasNext() is true
  System.out.print("Iterate from first node:");
  try {
    IRISIterator iter = dbnative.getIRISIterator("myNames","people");
    while (iter.hasNext()) {
      iter.next();
      if (iter.getValue()>=0) {
        System.out.print(" \"" + iter.getSubscriptValue() + "\"=" + iter.getValue()); }
      else {
        iter.remove();
      }

    };
  } catch  (Exception e) {
    System.out.println( e.getMessage());
  }
```

- The call to **getIRISIterator()** creates iterator instance *iter* for the immediate children of *^myNames("people")*.
- Each iteration of the `while` loop performs the following actions:
  - **next()** determines the subscript of the next valid node in collation order and positions the iterator at that node. (In the first iteration, the subscript is `"Anna"` and the node value is `2`).
  - If the node value returned by **getValue()** is negative, **remove()** is called to delete the node (including any subnodes. This is equivalent to calling **kill()** on the current node).

    Otherwise, **getSubscriptValue()** and **getValue()** are used to print the subscript and value of the current node.
- The `while` loop is terminated when **hasNext()** returns `false`, indicating that there are no more child nodes in this sequence.

This code prints the following line (element `"Zorro"` was not printed because its value was negative):

```
Iterate from first node: "Anna"=2 "Julia"=4 "Misha"=5 "Ruri"=3 "Vlad"=1
```

This example is extremely simple, and would fail in several situations. What if we don't want to start with the first or last node? What if the code attempts to get a value from a valueless node? What if the global array has data on more than one level? The following sections describe how to deal with these situations.

2.2.2 Finding Subnodes on All Levels

The next example will search a slightly more complex set of subnodes. We'll add new child node *"dogs"* to *^myNames* and use it as the target node for this example:

```
^myNames                                        (valueless root node)
   ^myNames("dogs")                             (valueless level 1 node)
      ^myNames("dogs","Balto") = 6
      ^myNames("dogs","Hachiko") = 8
      ^myNames("dogs","Lassie")                 (valueless level 2 node)
         ^myNames("dogs","Lassie","Timmy") = 10 (level 3 node)
      ^myNames("dogs","Whitefang") = 7
   ^myNames("people")                           (valueless level 1 node)
      [five child nodes]                        (as listed in previous example)
```

Target node *^myNames("dogs")* has five subnodes, but only four of them are child nodes. In addition to the four level 2 subnodes, there is also a level 3 subnode, *^myNames("dogs","Lassie","Timmy")*. The search will not find *"Timmy"* because this subnode is the child of *"Lassie"* (not *"dogs"*), and therefore is not a sibling of the others.

Note: **Subscript Lists and Node Levels**

The term *node level* refers to the number of subscripts in the subscript list. For example, *^myGlobal("a","b","c")* is a "level 3 node," which is just another way of saying "a node with three subscripts."

Although node *^myNames("dogs","Lassie")* has a child node, it does not have a value. A call to **getValue()** will return `null` in this case. The following example searches for children of *^myNames("dogs")* in reverse collation order:

Get nodes in reverse order from last node under ^*myNames("dogs")*

```
// Read child nodes in descending order while iter.next() is true
  System.out.print("Descend from last node:");
  try {
    IRISIterator iter = dbnative.getIRISIterator("myNames","dogs");
    while (iter.hasPrevious()) {
      iter.previous();
      System.out.print(" \"" + iter.getSubscriptValue() + "\"")
      if (iter.getValue()=null) set(^myNames("dogs",iter.getSubscriptValue()),0);
      System.out.print("=" + iter.getValue())
    };
  } catch  (Exception e) {
    System.out.println( e.getMessage());
  }
```

This code prints the following line:

```
Descend from last node: "Whitefang"=7 "Lassie" "Hachiko"=8 "Balto"=6
```

In the previous example, the search misses several of the nodes in global array ^*myNames* because the scope of the search is restricted in various ways:

- Node ^*myNames("dogs","Lassie","Timmy")* is not found because it is not a level 2 subnode of ^*myNames("dogs")*.
- Level 2 nodes under ^*myNames("people")* are not found because they are not siblings of the level 2 nodes under ^*myNames("dogs")*.

The problem in both cases is that **previous()** and **next()** only find nodes that are under the same parent and on the same level as the starting address. You must specify a different starting address for each group of sibling nodes.

In most cases, you will probably be processing a known structure, and will traverse the various levels with simple nested calls. In the less common case where a structure has an arbitrary number of levels, the following jdbc.IRIS method can be used to determine if a given node has subnodes:

- **isDefined()** — returns `0` if the specified node does not exist, `1` if the node exists and has a value. `10` if the node is valueless but has subnodes, or `11` if it has both a value and subnodes.

If **isDefined()** returns `10` or `11`, subnodes exist and can be processed by creating an iterator as described in the previous examples. A recursive algorithm could use this test to process any number of levels.

2.3 Transactions and Locking

The following topics are discussed in this section:

- Controlling Transactions — describes using methods of jdbc.IRIS 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.

Important: **Never Mix Transaction Models**

DO NOT mix the Native transaction model with the SQL (JDBC) transaction model.

- If you want to use only Native commands within a transaction, you should always use Native transaction methods.
- If you want to use a mix of Native and JDBC/SQL commands within a transaction, you should turn autoCommit OFF and then always use Native transaction methods.
- If you want to use only JDBC/SQL commands within a transaction, you can either always use SQL transaction methods, or turn autocommit OFF and then always use Native transaction methods.

2.3.1 Controlling Transactions

The jdbc.IRIS class provides the following methods to control transactions:

- **tCommit()** — commits one level of transaction.
- **tStart()** — starts a transaction (which may be a nested transaction).
- **getTLevel()** — returns an int value indicating the current transaction level (0 if not in a transaction).
- **tRollback()** — rolls back all open transactions in the session.
- **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";
  dbnative.tStart();
  dbnative.set("firstValue", globalName, dbnative.getTLevel());
  // getTLevel() is 1 and ^myGlobal(1) = "firstValue"

  dbnative.tStart();
  dbnative.set("secondValue", globalName, dbnative.getTLevel());
  // getTLevel() is 2 and ^myGlobal(2) = "secondValue"

  dbnative.tStart();
  dbnative.set("thirdValue", globalName, dbnative.getTLevel());
  // getTLevel() is 3 and ^myGlobal(3) = "thirdValue"

  System.out.println("Node values before rollback and commit:");
  for (int ii=1;ii<4;ii++) {
    System.out.print(globalName + "(" + ii + ") = ");
    if (dbnative.isDefined(globalName,ii) > 1)
System.out.println(dbnative.getString(globalName,ii));
    else System.out.println("<valueless>");
  }
// prints: Node values before rollback and commit:
//         ^myGlobal(1) = firstValue
//         ^myGlobal(2) = secondValue
//         ^myGlobal(3) = thirdValue

  dbnative.tRollbackOne();
  dbnative.tRollbackOne();  // roll back 2 levels to getTLevel 1
  dbnative.tCommit();  // getTLevel() after commit will be 0
  System.out.println("Node values after the transaction is committed:");
  for (int ii=1;ii<4;ii++) {
    System.out.print(globalName + "(" + ii + ") = ");
    if (dbnative.isDefined(globalName,ii) > 1)
System.out.println(dbnative.getString(globalName,ii));
    else System.out.println("<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 jdbc.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
```

- **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.
- **unlock()** — Takes *lockMode*, *globalName*, and *subscripts* arguments, and releases the lock on a node.

  Methods Connection.**close()** or jdbc.IRIS.**releaseAllLocks()** (see "Using Locks in a Transaction") will release all currently held locks.

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 **read(pressEnter)** in the following code will pause execution so that you can look at the list whenever it changes.

The following methods will release all currently held locks:

- jdbc.IRIS.**releaseAllLocks()** — releases all locks currently held by this connection.
- Connection.**close()** — releases all locks and other connection resources before it closes the connection.

The following examples demonstrate the various lock and release methods.

Using Locks in a Transaction: Using incremental locking

```
  dbnative.set("my node", "nodeRef1", "my-node");
  dbnative.set("shared node", "nodeRef2", "shared-node");
  byte[] pressEnter = new byte[4];
  try {
    dbnative.tStart();
// lock ^nodeRef1("my-node") exclusively
    dbnative.lock("E",10,"nodeRef1", "my-node");
// lock ^nodeRef2 shared
    dbnative.lock("ES",10,"nodeRef2", "shared-node");
    System.out.println("Exclusive lock on ^nodeRef1(\"my-node\") and shared lock on ^nodeRef2");

    System.out.println("Press return to release locks individually");
    System.in.read(pressEnter); // Wait for user to press Return

// release ^nodeRef1("my-node") after transaction
```

```
    dbnative.unlock("E",,"nodeRef1", "my-node");
// release ^nodeRef2 immediately
    dbnative.unlock("ES",,"nodeRef2", "shared-node");
    System.out.println("Press return to commit transaction");
    System.in.read(pressEnter);
    dbnative.tCommit();
  }
  catch (Exception e) { System.out.println(e.getMessage()); }
  catch (java.io.IOException e) { System.out.println(e.getMessage()); }
```

Using Locks in a Transaction: Using non-incremental locking

```
// lock ^nodeRef1("my-node") non-incremental
  dbnative.lock("",10,"nodeRef1", "my-node");
  System.out.println("Exclusive lock on ^nodeRef1(\"my-node\"), return to lock ^nodeRef1
non-incrementally");
  System.in.read(pressEnter);

// lock ^nodeRef2 shared non-incremental
  dbnative.lock("S",10,"nodeRef2", "shared-node");
  System.out.println("Verify that only ^nodeRef2 is now locked, then press return");
  System.in.read(pressEnter);
```

Using Locks in a Transaction: Using releaseAllLocks() to release all incremental locks

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

// lock ^nodeRef2 exclusive incremental
  dbnative.lock("E",10,"nodeRef2", "shared-node");
  System.out.println("Two locks are held (one with lock count 2), return to release both locks");

  System.in.read(pressEnter);

  dbnative.releaseAllLocks();
  System.out.println("Verify both locks have been released");
  System.in.read(pressEnter);
```

3

Calling ObjectScript Methods and Functions

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

Important: **Trailing Arguments**

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.

3.1 Class Method Calls

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

- **classMethodBoolean**
- **classMethodBytes**
- **classMethodDouble**
- **classMethodLong**
- **classMethodString**
- **classMethodVoid**

The following code demonstrates a method call for each datatype:

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

  // calling class methods
  System.out.println("Boolean class method: " + IRIS.classMethodBoolean(className,"Boolean",false));
  System.out.println("Bytes class method: " + new
String(IRIS.classMethodBytes(className,"Bytes","byteArray")));
  System.out.println("String class method: " + IRIS.classMethodString(className,"String","Java Test"));

  System.out.println("Long class method: " + IRIS.classMethodLong(className,"Long",7,8));
  System.out.println("Double class method: " + IRIS.classMethodDouble(className,"Double",7.56));
  IRIS.classMethodVoid(className,"Void",67);
  System.out.println("Void class method returned successfully.");
```

This example assumes that the following ObjectScript class (User.JavaTest) is compiled and available on the server:

```
Class User.JavaTest
{

  ClassMethod Void(p1 As %Integer)
  {
          Set ^p1=p1
          Quit
  }

  ClassMethod String(p1 As %String) As %String
  {
          Quit "Hello "_p1
  }

  ClassMethod Bytes(p1 As %String) As %Binary
  {
          Quit $C(65,66,67,68,69,70,71,72,73,74)
  }

  ClassMethod Long(p1 As %Integer, p2 As %Integer) As %Integer
  {
          Quit p1+p2
  }

  ClassMethod Double(p1 As %Double) As %Double
  {
          Quit p1 * 100
  }

  ClassMethod Boolean(p1 As %Boolean) As %Boolean
  {
          Quit 0
  }
}
```

3.2 Function Calls

The following methods call a specified ObjectScript function. They take String arguments for *functionName* and *routineName*, plus an Object containing `0` or more function arguments, and return the result as the indicated type:

- **functionBoolean**
- **functionBytes**
- **functionDouble**
- **functionLong**
- **functionString**
- **procedure**

The following code demonstrates a function call for each datatype:

```
  String routineName = "JavaTest";

  // calling functions
  System.out.println("Boolean function: " + IRIS.functionBoolean ("Boolean",routineName,false));
  System.out.println("Bytes function: " + new String(IRIS.functionBytes
("Bytes",routineName,"byteArray")));
  System.out.println("String function: " + IRIS.functionString ("String",routineName,"Java Test"));
  System.out.println("Long function: " + IRIS.functionLong ("Long",routineName,7,8));
  System.out.println("Double function: " + IRIS.functionDouble ("Double",routineName,7.56));
  IRIS.procedure("Procedure",routineName,67);
  System.out.println("Procedure returned successfully.");
```

This example assumes that the following ObjectScript routine (**JavaTest.mac**) is compiled and available on the server:

```
Procedure(p1) public {
    Set ^p1=p1
    Quit
}

String(p1) public {
    Quit "Hello "_p1
}

Bytes(p1) public {
    Quit $C(65,66,67,68,69,70,71,72,73,74)
}

Long(p1,p2) public {
    Quit p1+p2
}

Double(p1) public {
    Quit p1 * 100
}

Boolean(p1) public {
    Quit 0
}
```

4

Native API Quick Reference

This is a quick reference for the InterSystems IRIS Native API, which consists of the following classes in com.intersystems.jdbc:

- 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 on these classes, see "InterSystems IRIS JDBC Driver" in the online JavaDoc.

jdbc.IRIS methods by usage

Instances of jdbc.IRIS are created by IRIS.**createIRIS()**. The jdbc.IRIS class provides global value get and set accessors for datatypes Boolean, Short, Integer, Long, Double, String, byte[], Time, Date, Timestamp, InputStream, Reader, and Serializable. The remaining methods are listed below, organized by usage:

Global Iteration and Management

- **increment()** — Increments the value of a global node by the specified amount.
- **isDefined()** — Checks if a global node exists, and if it contains data.
- **kill()** — Kills the global node including any descendants.
- **getIRISIterator()** — Returns an IRISIterator instance.

Transactions and Locking

- **lock**
- **unlock**
- **releaseAllLocks**
- **getTLevel**
- **tCommit**
- **tRollback**
- **tRollbackOne**
- **tStart**

ClassMethod and Function Calls by Datatype

- **classMethodBoolean**, **functionBoolean**
- **classMethodBytes**, **functionBytes**
- **classMethodDouble**, **functionDouble**
- **classMethodLong**, **functionLong**
- **classMethodString**, **functionString**
- **classMethodVoid**, **procedure**

4.1 Class IRIS

For the most recent information on this class, see the IRIS section of "InterSystems IRIS JDBC Driver" in the online JavaDoc.

Instances of com.intersystems.jdbc.IRIS are created by calling IRIS.**createIRIS()**.

createIRIS()

Constructor jdbc.IRIS.**createIRIS()** returns an instance of jdbc.IRIS that uses the specified Connection.

```
static IRIS createIRIS(IRISConnection conn) throws SQLException
```

parameters:

- `conn` — an instance of IRISConnection.

classMethodBoolean()

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

```
final Boolean classMethodBoolean (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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

classMethodBytes()

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

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

classMethodDouble()

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

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

classMethodLong()

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

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

classMethodString()

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

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

functionBoolean()

jdbc.IRIS.**functionBoolean()** calls a function, passing 0 or more arguments and returning an instance of Boolean.

```
final Boolean functionBoolean (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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

functionBytes()

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

```
final 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 method arguments of supported types (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

functionDouble()

jdbc.IRIS.**functionDouble()** calls a function, passing 0 or more arguments and returning an instance of Double.

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

functionLong()

jdbc.IRIS.**functionLong()** calls a function, passing 0 or more arguments and returning an instance of Long.

```
final Long functionLong (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 (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

functionString()

jdbc.IRIS.**functionString()** calls a function, passing 0 or more arguments and returning an instance of String.

```
final 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 method arguments of supported types (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

getIRISIterator()

jdbc.IRIS.**getIRISIterator()** returns an IRISIterator object (see "Class IRISIterator") for the specified node.

```
final IRISIterator getIRISIterator(String globalName, Object... subscripts)
```

parameters:

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

getObject()

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

```
final Object getObject (String globalName, String... subscripts )
```

parameters:

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

getReader()

jdbc.IRIS.**getReader()** gets the value of the global as a java.io.Reader (or `null` if node does not exist). This method is currently limited to the maximum size of a single global node.

```
final Reader getReader (String globalName, String... subscripts )
```

parameters:

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

getShort()

jdbc.IRIS.**getShort()** gets the value of the global as a Short (or `null` if node does not exist). Returns 0 if node value is empty string.

```
final Short getShort (String globalName, String... subscripts )
```

parameters:

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

getString()

jdbc.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 Java is translated to the null string character `$CHAR(0)` in ObjectScript. A null in Java is translated to the empty string in ObjectScript. This translation is consistent with the way JDBC handles these values.

```
final String getString (String globalName, String... subscripts )
```

parameters:

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

getTime()

jdbc.IRIS.**getTime()** gets the value of the global as a java.sql.Time (or `null` if node does not exist).

```
final java.sql.Time getTime (String globalName, String... subscripts )
```

parameters:

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

getTLevel()

jdbc.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. *Note*: Never mix Native API and SQL transaction models (see the warning in "Transactions and Locking").

```
final Integer getTLevel ( )
```

increment()

jdbc.IRIS.**increment()** increments the specified global with the passed value. A null value is interpreted as 0. Returns the new value of the global node

```
final long increment (Integer value, String globalName, String... subscripts )
```

parameters:

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

isDefined()

jdbc.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.

```
final int isDefined (String globalName, String... subscripts )
```

parameters:

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

kill()

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

```
final void kill (String globalName, String... subscripts )
```

parameters:

- `globalName` — global name

- `subscripts` — array of subscripts for this node

lock()

jdbc.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 COS.

```
final boolean lock (String lockMode, Integer timeout, String globalName, String... 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

procedure()

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

```
final 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 procedure arguments of supported types (see "Calling ObjectScript Methods and Functions"). Trailing arguments may be omitted.

releaseAllLocks()

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

```
final void releaseAllLocks ( )
```

set()

jdbc.IRIS.**set()** sets the current node to a value of a supported datatype:

```
final void set (Boolean value, String globalName, String... subscripts )
final void set (Short value, String globalName, String... subscripts )
final void set (Integer value, String globalName, String... subscripts )
final void set (Long value, String globalName, String... subscripts )
final void set (Double value, String globalName, String... subscripts )
final void set (String value, String globalName, String... subscripts )
final void set (byte[] value, String globalName, String... subscripts )

final void set (java.sql.Date value, String globalName, String... subscripts )
final void set (java.sql.Time value, String globalName, String... subscripts )
final void set (java.sql.Timestamp value, String globalName, String... subscripts )
final void set (java.io.InputStream value, String globalName, String... subscripts )
final void set (java.io.Reader value, String globalName, String... subscripts )
final void set (Object value, String globalName, String... subscripts )
final<T extends Serializable> void set (T value, String globalName, String... subscripts )
```

parameters:

- `value` — value of a supported datatype (`null` value sets global to `""`).

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

Notes on specific datatypes

The following datatypes have some extra features:

- String — empty string and null values require some translation. An empty string `""` in Java is translated to the null string character `$CHAR(0)` in ObjectScript. A null in Java is translated to the empty string in ObjectScript. This translation is consistent with the way JDBC handles these values.
- java.io.InputStream — currently limited to the maximum size of a single global node.
- java.io.Reader — currently limited to the maximum size of a single global node.
- java.io.Serializable — value can be set to an instance of an object that implements Serializable. The Java object will be serialized prior to being set as the global value. Use **getObject()** to retrieve the value.

tCommit()

jdbc.IRIS.**tCommit()** commits the current transaction. *Note*: Never mix Native API and SQL transaction models (see the warning in "Transactions and Locking").

```
final void tCommit ( )
```

tRollback()

jdbc.IRIS.**tRollback()** rolls back all open transactions in the session. *Note*: Never mix Native API and SQL transaction models (see the warning in "Transactions and Locking").

```
final void tRollback ( )
```

tRollbackOne()

jdbc.IRIS.**tRollbackOne()** rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back. *Note*: Never mix Native API and SQL transaction models (see the warning in "Transactions and Locking").

```
final void tRollbackOne ( )
```

tStart()

jdbc.IRIS.**tStart()** starts/opens a transaction. *Note*: Never mix Native API and SQL transaction models (see the warning in "Transactions and Locking").

```
final void tStart ( )
```

Timestamp()

jdbc.IRIS.**Timestamp()** gets the value of the global as a java.sql.Timestamp (or `null` if node does not exist).

```
final java.sql.Timestamp getTimestamp (String globalName, String... subscripts )
```

parameters:

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

unlock()

jdbc.IRIS.**unlock()** unlocks the global. This method performs an incremental unlock, not the implicit unlock-before-lock feature that is also offered in ObjectScript.

```
final void unlock (String lockMode, String globalName, String... 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

classMethodVoid()

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

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

4.2 Class IRISIterator

For the most recent information on this class, see the IRISIterator section of "InterSystems IRIS JDBC Driver" in the online JavaDoc.

Instances of com.intersystems.jdbc.IRISIterator are created by calling jdbc.IRIS.**getIRISIterator()**. See "Finding Nodes in a Global Array" for more details and examples.

getSubscriptValue()

IRISIterator.**getSubscriptValue()** returns a string containing the current subscript. Throws IllegalStateException if **remove()** has been called on the current node with this iterator, or there is no last element returned by this iterator (i.e. no successful **next()** or **previous()** calls).

```
String  getSubscriptValue () throws IllegalStateException
```

getValue()

IRISIterator.**getValue()** returns the value of the current node. Throws IllegalStateException if **remove()** has been called on the current node with this iterator, or there is no last element returned by this iterator (i.e. no successful **next()** or **previous()** calls).

```
Object  getValue () throws IllegalStateException
```

hasNext()

IRISIterator.**hasNext()** returns true if the iteration has more elements. (In other words, returns true if **next()** would return an element rather than throwing an exception.)

```
boolean  hasNext ()
```

hasPrevious()

IRISIterator.**hasPrevious()** returns true if the iteration has a previous element. (In other words, returns true if **previous()** would return an element rather than throwing an exception.)

```
boolean  hasPrevious ()
```

next()

IRISIterator.**next()** returns the next element in the iteration. Throws NoSuchElementException if the iteration has no more elements

```
String  next () throws NoSuchElementException
```

previous()

IRISIterator.**previous()** returns the previous element in the iteration. Throws NoSuchElementException if the iteration does not have a previous element

```
String  previous () throws NoSuchElementException
```

remove()

IRISIterator.**remove()** removes from the underlying collection the last element returned by this iterator. This method can be called only once per call to **next()** or **previous()**. Throws IllegalStateException if **remove()** has been called on the current node with this iterator, or there is no last element returned by this iterator (i.e. no successful **next()** or **previous()** calls).

```
void  remove () throws IllegalStateException
```

startFrom()

IRISIterator.**startFrom()** sets the iterator's starting position to the specified subscript. The starting position does not have to be a valid sub-node.

```
void  startFrom(Object subscript)
```

After calling this method, use **next()** or **previous()** to advance the iterator to the next defined sub-node in alphabetic collating sequence. The iterator will not be positioned at a defined sub-node until one of these methods is called. If you call **getSubscriptValue()** or **getValue()** before the iterator is advanced, an IllegalStateException will be thrown.