InterSystems IRIS Data Platform

Using the Native API for Java

Version 2020.3
2021-02-04

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

Using the Native API for Java
InterSystems IRIS Data Platform   Version 2020.3   2021-02-04
Copyright © 2021 InterSystems Corporation
All rights reserved.

Table of Contents

List of Figures

About This Book

The InterSystems *Native API for Java* is a lightweight extension to the InterSystems JDBC driver. It gives your Java application direct access to powerful InterSystems IRIS® resources formerly available only in ObjectScript. You can call ObjectScript methods and functions, create and control instances of an ObjectScript class, directly access InterSystems global arrays, and much more. All Native API classes access the database through a standard JDBC connection, and can be used without any special setup or installation procedures.

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

- Introduction to the Native API — gives an overview of Native API abilities and provides some simple code examples.
- Calling ObjectScript Methods and Functions — describes how to call ObjectScript classmethods and functions.
- Using Java Reverse Proxy Objects — demonstrates how to manipulate ObjectScript objects through Object Gateway reverse proxy objects.
- 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.
- Transactions and Locking — describes how to work with the Native API transaction and concurrency control model.
- Native API Quick Reference — provides a brief description of each Native API method mentioned in this book.

There is also a detailed Table of Contents.

More information about globals

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

- *Using the Native API for .NET*
- *Using the Native API for Python*
- *Using the Native API for Node.js*

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

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

InterSystems Core APIs for Java

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

- *Using Java with InterSystems Software* — provides an overview of all InterSystems Java technologies, and describes how use the InterSystems JDBC driver for relational data access to external data sources.
- *Persisting Java Objects with XEP* — describes how to store and retrieve Java objects using the InterSystems XEP API.

1 Introduction to the Native API

The *Native API for Java* is a lightweight interface to powerful InterSystems IRIS® resources that were once available only through ObjectScript. All Native API classes are part of the InterSystems JDBC driver (com.intersystems.jdbc). They access the database through a standard JDBC connection, and can be used without any special setup or installation procedures. The Native API is a tool with many uses:

- Call ObjectScript classmethods and functions — Access any available ObjectScript class or routine (including the huge collection in the Class Library) and call ObjectScript classmethods or functions from your Java application as easily as you can call native Java methods.
- Directly control instances of an ObjectScript class — Create a Java proxy object to control an object on the server. The proxy can be used to call instance methods and get or set properties just as if you were using the server object directly.
- Work with multidimensional global arrays — Gain direct access to the high performance native data structures. Global arrays can be created, read, changed, and deleted in Java applications just as they can in ObjectScript.

The following brief examples demonstrate how easy it is to add these abilities to your Java application.

Call ObjectScript methods and functions

Your Java application can call any ObjectScript method or function as easily as it calls native Java methods (see "Calling ObjectScript Methods and Functions").

```
String currentNameSpace = iris.classMethodString("%SYSTEM.SYS","NameSpace");
```

This example calls one of the many methods available from the InterSystems Class Library, but you can just as easily access your own custom ObjectScript code. The following example uses a similar method to create a new instance of an ObjectScript class.

Create and use individual instances of an ObjectScript class

Your application can create an instance of an ObjectScript class, immediately generate a Java proxy object for it, and use the proxy to work with the ObjectScript instance (see "Using Java Reverse Proxy Objects").

In this example, the first line calls the **%New()** method of ObjectScript class Demo.dataStore, creating an instance in InterSystems IRIS. In Java, the call returns a corresponding proxy object named *dataStoreProxy*, which is used to call instance methods and get or set properties of the ObjectScript instance:

```
// use a classmethod call to create an ObjectScript instance and generate a proxy object
  IRISObject dataStoreProxy = (IRISObject)iris.classMethodObject("Demo.dataStore","%New");

// use the proxy to call instance methods, get and set properties
  dataStoreProxy.invokeVoid("ititialize");
  dataStoreProxy.set("propertyOne","a string property");
  String testString = dataStoreProxy.get("propertyOne");
  dataStoreProxy.invoke("updateLog","PropertyOne value changed to "+testString);

// pass the proxy back to ObjectScript method ReadDataStore()
  iris.classMethodObject("Demo.useData","ReadDataStore",dataStoreProxy);
```

The last line of this example passes the *dataStoreProxy* proxy to an ObjectScript method named *ReadDataStore()*, which interprets it as a reference to the original ObjectScript instance. From there, the instance could be saved to the database, passed to another ObjectScript application, or even passed back to your Java application.

Work with multidimensional global arrays

The Native API provides all the methods needed to manipulate global arrays (see "Working with Global Arrays"). You can easily access and manipulate globals, traverse multilevel global arrays, and inspect data structures just as you can in ObjectScript. The following example demonstrates how to create, read, change, and delete a simple global array.

```
// Create a global (ObjectScript equivalent: set ^myGlobal("subOne") = 10)
  iris.set(10, "myGlobal", "subOne");

// Change, read, and delete the global
  iris.increment(2, "myGlobal", "subOne")   // increment value to 12
  System.out.print("New number is " + iris.getInteger("myGlobal", "subOne"));
  iris.kill("myGlobal", "subOne");
```

2
Calling ObjectScript Methods and Functions

This chapter describes methods of class IRIS that allow you to call ObjectScript class methods and user-defined functions directly from your Java application. See the following sections for details and examples:

- Class Method Calls — demonstrates how to call ObjectScript class methods.
- Function Calls — demonstrates how to call user defined ObjectScript functions and procedures.
- Calling Class Library Methods — demonstrates how to pass arguments by reference and check %Status codes.

2.1 Class Method Calls

The following methods of class IRIS call ObjectScript class methods, returning values of the type indicated by the method name: **classMethodBoolean()**, **classMethodBytes()**, **classMethodDouble()**, **classMethodIRISList()**, **classMethodLong()**, **classMethodObject()**, **classMethodString()**, and **classMethodVoid()**. You can also use **classMethodStatusCode()** to retrieve error messages from class methods that return ObjectScript %Status (see "Catching %Status Error Codes").

All of these methods take String arguments for *className* and *methodName*, plus `0` or more method arguments, which may be any of the following types: Integer, Short, String, Long, Double, Float, byte[], Boolean, Time, Date, Timestamp, IRISList, or IRISObject. If the connection is bidirectional (see "Using Java Reverse Proxy Objects"), then any Java object can be used as an argument.

Trailing arguments may be omitted in argument lists, 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.

Calling ObjectScript class methods

The code in this example calls class methods of each supported datatype from ObjectScript test class User.NativeTest (listed immediately after the example). Assume that variable *iris* is a previously defined instance of class IRIS and is currently connected to the server.

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

comment = "cmBoolean() tests whether two numbers are equal (true=1,false=0): ";
boolean boolVal = iris.classMethodBoolean(className,"cmBoolean",7,7);
System.out.println(comment+boolVal);
```

```
comment = "cmBytes creates byte array [72,105,33]. String value of array:: ";
byte[] byteVal = iris.classMethodBytes(className,"cmBytes",72,105,33);
System.out.println(comment+(new String(byteVal)));

comment = "cmString() concatenates \"Hello\" + arg: ";
String stringVal = iris.classMethodString(className,"cmString","World");
System.out.println(comment+stringVal);

comment = "cmLong() returns the sum of two numbers: ";
Long longVal = iris.classMethodLong(className,"cmLong",7,8);
System.out.println(comment+longVal);

comment = "cmDouble() multiplies a number by 1.5: ";
Double doubleVal = iris.classMethodDouble(className,"cmDouble",10);
System.out.println(comment+doubleVal);

comment = "cmProcedure assigns a value to global array ^cmGlobal: ";
iris.classMethodVoid(className,"cmVoid",67);
// Read global array ^cmGlobal and then delete it
System.out.println(comment+iris.getInteger("^cmGlobal"));
iris.kill("cmGlobal");

comment = "cmList() returns a $LIST containing two values: ";
IRISList listVal = iris.classMethodList(className,"cmList","The answer is ",42);
System.out.println(comment+listVal.get(1)+listVal.get(2));
```

ObjectScript Class User.NativeTest

To run the previous example, this ObjectScript class must be compiled and available on the server:

```
Class User.NativeTest Extends %Persistent
{
ClassMethod cmBoolean(cm1 As %Integer, cm2 As %Integer) As %Boolean
{
    Quit (cm1=cm2)
}
ClassMethod cmBytes(cm1 As %Integer, cm2 As %Integer, cm3 As %Integer) As %Binary
{
    Quit $CHAR(cm1,cm2,cm3)
}
ClassMethod cmString(cm1 As %String) As %String
{
    Quit "Hello "_cm1
}
ClassMethod cmLong(cm1 As %Integer, cm2 As %Integer) As %Integer
{
    Quit cm1+cm2
}
ClassMethod cmDouble(cm1 As %Double) As %Double
{
    Quit cm1 * 1.5
}
ClassMethod cmVoid(cm1 As %Integer)
{
    Set ^cmGlobal=cm1
    Quit ^cmGlobal
}
ClassMethod cmList(cm1 As %String, cm2 As %Integer)
{
    Set list = $LISTBUILD(cm1,cm2)
    Quit list
}
}
```

You can test these methods by calling them from the Terminal. For example:

```
USER>write ##class(User.NativeTest).cmString("World")
Hello World
```

2.2 Function Calls

Function calls are similar to method calls, but the arguments are in a different order. The function label is specified first, followed by the name of the routine that contains it. This corresponds to the order used in ObjectScript, where function calls have the following form :

```
set result = $$myFunctionLabel^myRoutineName([arguments])
```

Functions are supported because they are necessary for older code bases (see “Callable User-defined Code Modules” in *Using ObjectScript*), but new code should always use method calls if possible.

The Native API methods in this section call user-defined ObjectScript functions or procedures and return a value of the type indicated by the method name: **functionBoolean()**, **functionBytes()**, **functionDouble()**, **functionIRISList()**, **functionObject()**, **functionLong()**, **functionString()**, or **procedure()** (no return value).

They take String arguments for *functionLabel* and *routineName*, plus `0` or more function arguments, which may be any of the following types: Integer, Short, String, Long, Double, Float, byte[], Boolean, Time, Date, Timestamp, IRISList, or IRISObject. If the connection is bidirection (see Using Java Reverse Proxy Objects), then any Java object can be used as an argument.

Trailing arguments may be omitted in argument lists, 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.

Note: **Built-in system functions are not supported**

These methods are designed to call functions in user-defined routines. ObjectScript system functions (which start with a `$` character. See “ObjectScript Functions” in the *ObjectScript Reference*) cannot be called directly from your Java code. However, you can call a system function indirectly by writing an ObjectScript wrapper function that calls the system function and returns the result. For example, the **fnList()** function (at the end of this section in ObjectScript Routine NativeRoutine.mac) calls $LISTBUILD.

Calling functions of ObjectScript routines with the Native API

The code in this example calls functions of each supported datatype from ObjectScript routine NativeRoutine (File NativeRoutine.mac, listed immediately after this example). Assume that *iris* is an existing instance of class IRIS, and is currently connected to the server.

```
String routineName = "NativeRoutine";
String comment = "";

comment = "fnBoolean() tests whether two numbers are equal (true=1,false=0): ";
boolean boolVal = iris.functionBool("fnBoolean",routineName,7,7);
System.out.println(comment+boolVal);

comment = "fnBytes creates byte array [72,105,33]. String value of the array: ";
byte[] byteVal = new String(iris.functionBytes("fnBytes",routineName,72,105,33));
System.out.println(comment+(new String(byteVal)));

comment = "fnString() concatenates \"Hello\" + arg: ";
String stringVal = iris.functionString("fnString",routineName,"World");
System.out.println(comment+stringVal);

comment = "fnLong() returns the sum of two numbers: ";
Long longVal = iris.functionInt("fnLong",routineName,7,8);
System.out.println(comment+longVal);

comment = "fnDouble() multiplies a number by 1.5: ";
Double doubleVal = iris.functionDouble("fnDouble",routineName,5);
System.out.println(comment+doubleVal);

comment = "fnProcedure assigns a value to global array ^fnGlobal: ";
iris.procedure("fnProcedure",routineName,88);
// Read global array ^fnGlobal and then delete it
System.out.println(comment+iris.getInteger("^fnGlobal")+"\n\n");
iris.kill("fnGlobal");

comment = "fnList() returns a $LIST containing two values: ";
IRISList listVal = iris.functionList("fnList",routineName,"The answer is ",42);
System.out.println(comment+listVal.get(1)+listVal.get(2));
```

ObjectScript Routine NativeRoutine.mac

To run the previous example, this ObjectScript routine must be compiled and available on the server:

```
fnBoolean(fn1,fn2) public {
   quit (fn1=fn2)
}
fnBytes(fn1,fn2,fn3) public {
    quit $CHAR(fn1,fn2,fn3)
}
fnString(fn1) public {
    quit "Hello "_fn1
}
fnLong(fn1,fn2) public {
    quit fn1+fn2
}
fnDouble(fn1) public {
    quit fn1 * 1.5
}
fnProcedure(fn1) public {
    set ^fnGlobal=fn1
    quit
}
fnList(fn1,fn2) public {
    set list = $LISTBUILD(fn1,fn2)
    quit list
}
```

You can test these functions by calling them from the Terminal. For example:

```
USER>write $$fnString^NativeRoutine("World")
Hello World
```

2.3 Calling Class Library Methods

Most of the classes in the InterSystems Class Library use a calling convention where methods only return a %Status value. The actual results are returned in arguments passed by reference. This section describes how to pass by reference and read %Status values.

- Using Pass-by-reference Arguments — demonstrates how to use the IRISReference class to pass objects by reference.
- Catching %Status Error Codes — describes how to use the **classMethodStatusCode()** method to test and read %Status values..

2.3.1 Using Pass-by-reference Arguments

The Native API supports pass by reference for both methods and functions. To pass an argument by reference, assign the argument value to an instance of class IRISReference and pass that instance as the argument:

```
  IRISReference valueRef = new IRISReference(""); // set inital value to null string
  iris.classMethodString("%SomeClass","SomeMethod",valueRef);
  String myString = valueRef.value;  // get the method result
```

Here is a working example:

Using pass-by-reference arguments

This example calls %SYS.DatabaseQuery.**GetDatabaseFreeSpace()** to get the amount of free space (in MB) available in the iristemp database.

```
  IRISReference freeMB = new IRISReference(0); // set inital value to 0
  String dir = "C:/InterSystems/IRIS/mgr/iristemp"; // directory to be tested
  Object status = null;

  try {
    System.out.print("\n\nCalling %SYS.DatabaseQuery.GetDatabaseFreeSpace()... ");
    status = iris.classMethodObject("%SYS.DatabaseQuery","GetDatabaseFreeSpace",dir,freeMB);
    System.out.println("\nFree space in " + dir + " = " + freeMB.value + "MB");
  }
  catch (RuntimeException e) {
    System.out.print("Call to class method GetDatabaseFreeSpace() returned error:");
    System.out.println(e.getMessage());
  }
```

prints:

```
Calling %SYS.DatabaseQuery.GetDatabaseFreeSpace()...
Free space in C:/InterSystems/IRIS/mgr/iristemp = 8.9MB
```

2.3.2 Catching %Status Error Codes

When a class method has ObjectScript %Status as the return type, you can use **classMethodStatusCode()** to retrieve error messages. When a class method call fails, the resulting RuntimeException error will contain the %Status error code and message.

In the following example, the **ValidatePassword()** method returns a %Status object. If the password is invalid (for example, password is too short) an exception will be thrown and the %Status message will explain why it failed. Assume that variable *iris* is a previously defined instance of class IRIS and is currently connected to the server.

Using classMethodStatusCode() to catch ObjectScript %Status values

This example passes an invalid password to **%SYSTEM.Security.ValidatePassword()** and catches the error message.

```
  String className = "%SYSTEM.Security";
  String methodName = "ValidatePassword";
  String pwd = ""; // an invalid password
  try {
// This call will throw a RuntimeException containing the %Status error message:
    iris.classMethodStatusCode(className,methodName,pwd);
// This call would fail silently or throw a generic error message:
    Object status = iris.classMethodObject(className,methodName,pwd);
    System.out.println("\nPassword validated!");

  } catch (RuntimeException e) {
    System.out.println("Call to "+methodName+"(\""+pwd+"\") returned error:");
    System.out.println(e.getMessage());
  }
```

Notice that this example deliberately calls a method that does not use any pass by reference arguments.

To experiment with a more complex example, you can try catching the status code in the previous example (Using pass-by-reference arguments). Force an exception by passing an invalid directory.

Note: **Using IRISObject.invokeStatusCode() when calling Instance methods**

The **classMethodStatusCode()** method is used for class method calls. When you are invoking proxy object instance methods (see "Using Java Reverse Proxy Objects") the IRISObject.**invokeStatusCode()** method can be used in exactly the same way.

3 Using Java Reverse Proxy Objects

The Native API works together with the Object Gateway for Java. *Reverse proxy objects* are Java objects that allow you to manipulate ObjectScript objects via the Object Gateway. You can use a reverse proxy object to call the target's instance methods and get or set property values, manipulating the ObjectScript target object as easily as if it were a native Java object.

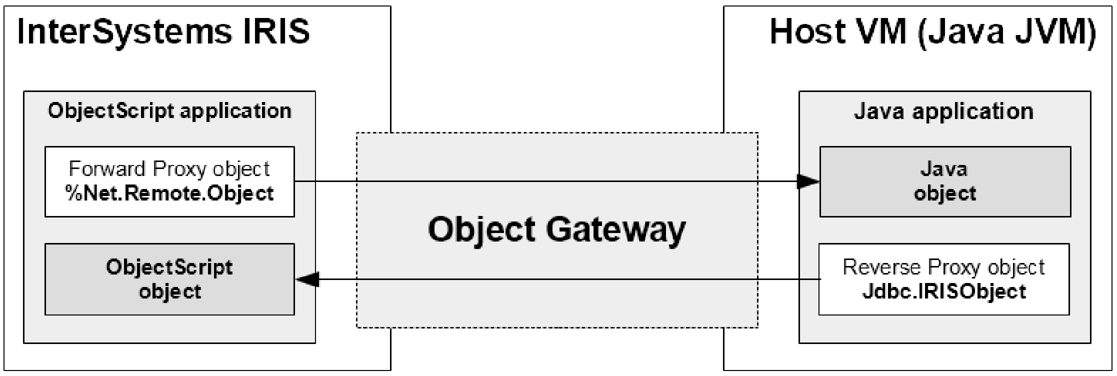
This chapter covers the following topics:

- Introducing the Object Gateway — provides a brief overview of the Object Gateway.
- Creating Reverse Proxy Objects — describes methods used to create reverse proxy objects.
- Using IRISObject Methods — demonstrates how reverse proxy objects are used.
- IRIS Object Supported Datatypes — describes datatype-specific versions of the reverse proxy methods.

3.1 Introducing the Object Gateway

The Object Gateway allows InterSystems IRIS objects and Java objects to interact freely, using the same connection and working together in the same context (database, session, and transaction). Object Gateway architecture is described in detail in *Using the Object Gateway for Java*, but for the purposes of this discussion you can think of it as a simple black box connecting proxy objects on one side to target objects on the other:

Figure 3–1: Object Gateway System



The connection is via TCP/IP if InterSystems IRIS and the host VM are on two different machines. If they are on the same machine, the connection can be either TCP/IP or shared memory.

As the diagram shows, a *forward proxy object* is an ObjectScript proxy for a Java target object (see "Using Dynamic Object Proxies" in *Using the Object Gateway for Java* for details). Forward proxies will be mentioned again later, but the focus of this chapter is how to use a Java reverse proxy to manipulate an ObjectScript target object.

3.2 Creating Reverse Proxy Objects

You can create a reverse proxy object by obtaining the OREF of an ObjectScript class instance (for example, by calling the **%New()** method of the class) and casting it to IRISObject. The following methods can be used to call ObjectScript class methods and functions:

- jdbc.IRIS.**classMethodObject()** calls an ObjectScript class method and returns the result as an instance of Object.
- jdbc.IRIS.**functionObject()** calls an ObjectScript function and returns the result as an instance of Object.

In either case, if the returned object is a valid OREF, it will be used to generate and return a proxy for the referenced object.

See "Class Method Calls" in the previous chapter for more information on how to call ObjectScript class methods. The following example uses **classMethodObject()** to create a reverse proxy object:

Creating an instance of IRISObject

- **classMethodObject()** is used to call the **%New()** method of an ObjectScript class named Demo.Test.
- Since the return value of **%New()** is a valid OREF, **classMethodObject()** creates and returns a proxy for the instance.
- In Java, the proxy is cast to IRISObject, creating reverse proxy object *test*:

```
IRISObject test = (IRISObject)iris.classMethodObject("Demo.Test","%New");
```

Variable *test* is a Java reverse proxy object for the new instance of Demo.Test. In the following section, *test* will be used to access methods and properties of the ObjectScript Demo.Test instance.

3.3 Using IRISObject Methods

IRISObject provides methods to access the ObjectScript target object. The following example uses **invoke()** and **invokeVoid()** to call instance methods, and accessors **get()** and **set()** to get and set a property value.

The example in this section uses reverse proxy object methods to access an ObjectScript class named Demo.Test, which includes declarations for methods **initialize()** and **add()**, and property *name*:

ObjectScript Class Demo.Test

```
Class Demo.Test Extends %Persistent
   Method initialize(initialVal As %String)
   Method add(val1 As %Integer, val2 As %Integer) As %Integer
   Property name As %String
```

In the following example, the first line creates a reverse proxy object named *test* for an instance of Demo.Test. The rest of the code uses proxy object *test* to access the Demo.Test instance.

Java code using a Demo.Test reverse proxy object

```
// Create an instance of Demo.Test and return a proxy object for it
  IRISObject test = (IRISObject)iris.classMethodObject("Demo.Test","%New");

// instance method test.initialize() is called with one argument, returning nothing.
  test.invokeVoid("initialize", 42);  // sets a mysterious internal variable to 42

// instance method test.add() is called with two arguments, returning an int value.
  int sum = test.invoke("add",2,3);  // adds 2 plus 3, returning 5

// The value of property test.name is set and then returned.
  test.set("name", "Einstein, Albert");  // sets the property to "Einstein, Albert"
  String name = test.get("name");   // returns the new property value
```

This example used the following IRISObject methods to access methods and properties of the Demo.Test instance:

- **classMethodObject()** calls Demo.Test class method **%New()**, which creates an instance of Demo.Test and returns an IRISObject proxy named *test* (as described previously in "Creating Reverse Proxy Objects").
- **invokeVoid()** invokes the **initialize()** instance method, which initializes an internal variable but does not return a value.
- **invoke()** invokes the **add()** instance method, which accepts two integer arguments and returns the sum as an integer.
- **set()** sets the *name* property to a new value.
- **get()** returns the value of property *name*.

There are also datatype-specific versions of these methods, as described in the following section.

3.4 IRISObject Supported Datatypes

The example in the previous section used the generic **set()**, **get()**, and **invoke()** methods, but the IRISObject class also provides datatype-specific methods for other supported datatypes.

IRISObject set() and get() methods

The IRISObject.**set()** method accepts any Java object as a property value, including all datatypes supported by IRIS.**set()** (see "Class IRIS Supported Datatypes").

In addition to the generic **get()** method, IRISObject provides the following type-specific methods: **getBoolean()**, **getBytes()**, **getDouble()**, **getIRISList()**, **getLong()**, **getObject(), getString()**, and **invokeVoid()**.

IRISObject invoke() methods

The IRISObject invoke methods support the same set of datatypes as the IRIS classmethod calls (see "Class Method Calls").

In addition to the generic **invoke()** method, IRISObject provides the following type-specific methods: **invokeBoolean()**, **invokeBytes()**, **invokeDouble()**, **invokeIRISList()**, **invokeLong()**, **invokeObject()**, **invokeString()**, and **invokeVoid()**. It also provides **invokeStatusCode()**, which gets the contents of an ObjectScript **%Status** return value (see "Catching %Status Error Codes").

All of the invoke methods take a String argument for *methodName* plus `0` or more method arguments, which may be any of the following types: Integer, Short, String, Long, Double, Float, byte[], Boolean, Time, Date, Timestamp, IRISList or IRISObject. If the connection is bidirectional, any Java object can be used as an argument.

Trailing arguments may be omitted in argument lists, 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.

4 Working with Global Arrays

This chapter covers the following topics:

- Introduction to Global Arrays — introduces global array concepts and provides a simple demonstration of how the Native API is used.
- Creating, Accessing, and Deleting Nodes — demonstrates how to create, change, or delete nodes in a global array, and how to retrieve node values.
- Finding Nodes in a Global Array — describes the iteration methods that allow rapid access to the nodes of a global array.
- Class IRIS Supported Datatypes — provides details on how to retrieve node values as specific datatypes.

Note: **IRIS Connection Code**

The examples in this chapter assume that an IRIS object named *iris* already exists and is connected to the server. The following code establishes a standard JDBC connection and creates an instance of IRIS:

```
//Open a connection to the server and create an IRIS object
      String connStr = "jdbc:IRIS://127.0.0.1:1972/USER";
      String user = "_SYSTEM";
      String pwd = "SYS";
      IRISConnection conn = (IRISConnection)
java.sql.DriverManager.getConnection(connStr,user,pwd);
      IRIS iris = IRIS.createIRIS(conn);
```

For more information on how to create an instance of IRIS, see the Quick Reference entry for **createIRIS()**. For general information on creating JDBC connections, see Establishing JDBC Connections in *Using Java with InterSystems Software*. .

4.1 Introduction to Global Arrays

A global array, like all sparse arrays, is a tree structure rather than a sequential 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 six 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:

```
  irisObject.Set("A", "root", "foo", "SubFoo");
  irisObject.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, Accessing, and Deleting Nodes".

The Native API code to create this array is demonstrated in the following example. An IRISConnection object establishes a connection to the server. The connection will be used by an instance of class IRIS named *iris*. Native API 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

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

public class NativeDemo {
  public static void main(String[] args) throws Exception {
    try {

//Open a connection to the server and create an IRIS object
      String connStr = "jdbc:IRIS://127.0.0.1:1972/USER";
      String user = "_SYSTEM";
      String password = "SYS";
      IRISConnection conn = (IRISConnection)
java.sql.DriverManager.getConnection(connStr,user,password);
      IRIS iris = IRIS.createIRIS(conn);

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

// Read the values from the database and print them
      String subfoo = iris.getString("root", "foo", "SubFoo");
      String underbar = iris.getString("root", "bar", "lowbar", "UnderBar");
      System.out.println("Created two values: \n"
         + " root(\"foo\",\"SubFoo\")=" + subfoo + "\n"
        + " root(\"bar\",\"lowbar\",\"UnderBar\")=" + underbar);

//Delete the global array and terminate
      iris.kill("root"); // delete global array root
      iris.close();
      conn.close();
    }
    catch (Exception e) {
      System.out.println(e.Message);
    }
  }// end main()
} // end class NativeDemo
```

NativeDemo prints the following lines:

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

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

- IRIS.**createIRIS()** creates a new instance of IRIS named *iris*, which will access the database through *conn*.

- IRIS.**set()** creates new persistent nodes in the database.
- IRIS.**getString()** queries the database and returns the values of the specified nodes.
- IRIS.**kill()** deletes the specified node and all of its subnodes from the database.

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

4.1.1 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 only exist as pointers to lower level nodes.

4.1.2 Global 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 or a number. String subscripts are case-sensitive and can use all characters (including control and non-printing characters). Length is limited only by the 511 character maximum for the total node address.

4.2 Creating, Accessing, and Deleting Nodes

The Native API provides three methods that can make changes in the database: **set()** and **increment()** can create nodes or change node values, and **kill()** can delete a node or set of nodes. Node values are retrieved by type-specific getter methods such as **getInteger()** and **getString()**.

- Creating Nodes and Setting Node Values — describes how to use **set()** and **increment()**.
- Getting Node Values — lists getter methods for each supported datatype.
- Deleting Nodes — describes how to use **kill()**.

4.2.1 Creating Nodes and Setting Node Values

The **set()** and **increment()** methods can be used to create a persistent node with a specified value, or to change the value of an existing node.

IRIS.**set()** takes a *value* argument of any supported datatype and stores the value at the specified address. If no node exists at the target address, a new one is created.

Setting and changing node values

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`.

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

set() can create and change values of any supported datatype. To read an existing value, you must use a different getter method for each datatype, as described in the next section.

IRIS.**increment()** takes a *number* argument, increments the node value by that amount, and returns the incremented value. The *number* argument 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. This method uses a thread-safe atomic operation to change the value of the node, so the node is never locked.

Incrementing node values

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

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

Note: **Global naming rules**

The second argument for either **set()** or **increment()** is a global array name. The name can include letters, numbers, and periods. It must begin with a character, and may not end with a period. The arguments after the global name are subscripts, which can be either numbers or strings (case-sensitive, not restricted to alphanumeric characters). See “Global Naming Rules” for more information.

4.2.2 Getting Node Values

The **set()** method can be used with all supported datatypes, but each datatype requires a separate getter. Node values can be any of the following datatypes: Boolean, byte[], Double, Float, Integer, Long, Short, String, Date, Time, Timestamp, plus Object, IRISList, subclasses of java.io.InputStream and java.io.Reader, and objects that implement java.io.Serializable.

The following methods are used to retrieve node values of these datatypes:

- Numeric datatypes: **getBoolean()**, **getShort()**, **getInteger()**, **getLong()**, **getDouble()**, **getFloat()**
- String and Binary datatypes: **getBytes()**, **getString()**
- Object and $list datatypes: **getObject()**, **getIRISList()**
- Temporal datatypes: **getDate()**, **getTime()**, **getTimestamp()**
- Other datatypes: **getInputStream()**, **getReader()**

For more information on datatypes, see “Class IRIS Supported Datatypes” later in this chapter.

4.2.3 Deleting 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.

In the following example, 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
```

The example will delete the entire global array by calling **kill()** on two of its subnodes, *myGlobal("A")* and *myGlobal("B",1)*.

Deleting a node or group of nodes

The first call will delete node *myGlobal("A")* and both of its subnodes:

```
iris.kill("myGlobal", "A");
// also kills child nodes myGlobal("A",1) and myGlobal("A",2)
```

The second call deletes *myGlobal("B",1)*, the last remaining subnode with a value:

```
iris.kill("myGlobal","B",1);
```

Since neither of the remaining nodes has a value, the entire global array is deleted:

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

4.3 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.

4.3.1 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.

4.3.2 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.

4.4 Class IRIS Supported Datatypes

For simplicity, examples in previous sections of this chapter have always used Integer or String node values, but the IRIS class also provides datatype-specific methods for the following supported datatypes.

IRIS.set()

The IRIS.**set()** method supports datatypes Boolean, byte[], Double, Integer, Long, Short, Float, String, IRISList, plus Java classes Date, Time, Timestamp, InputStream, Reader, and classes that implement Serializable. A `null` value is stored as `""`.

Class IRIS getters for numeric values

The following IRIS methods assume that the node value is numeric, and attempt to convert it to an appropriate Java variable: **getBoolean()**, **getShort()**, **getInteger()**, **getLong()**, **getDouble()**, or **getFloat()**. The numeric fetch methods will throw UndefinedException if the target node is valueless or does not exist.

Given an Integer node value, all numeric methods return meaningful values. The **getInteger()** and **getLong()** methods cannot be applied to Double or Float values with reliable results, and may throw an exception for these values.

Class IRIS getters for String, byte[], and IRISList

In the InterSystems IRIS database, String, byte[], and IRISList objects are all stored as strings, and no information about the original datatype is preserved. The IRIS **getString()**, **getBytes()**, and **getIRISList()** methods get string data and return it in the desired format.

The string getters assume that the node value is non-numeric, and attempt to convert it appropriately. They return `null` if the target node is valueless or does not exist. These methods do not perform any type checking, and will not usually throw an exception if the node value is of the wrong datatype.

Class IRIS getters for Java classes

The IRIS class also supports getters for Java classes Date, Time, Timestamp, InputStream, and Reader. Classes that implement Serializable can be retrieved with **getObject()**.

- **getDate()**, **getTime()**, **getTimestamp()** — get java.sql datatypes Date, Time, and Timestamp.
- **getInputStream()** — gets objects that implement java.io.InputStream.
- **getReader()** — gets objects that implement java.io.Reader.
- **getObject()** — gets the value of the target node and returns it as an Object.

  Objects that implement java.io.Serializable can be retrieved by casting the return value of **getObject()** to the appropriate class.

Important: **Getter methods do not check for incompatible datatypes**

These methods are optimized for speed, and never perform type checking. Your application should never depend on an exception being thrown if one of these methods attempts to fetch a value of the wrong datatype. Although an exception may be thrown, it is more likely that the method will fail silently, returning an inaccurate or meaningless value.

5 Transactions and Locking

The *Native API for Java* offers an alternative to the standard java.sql transaction model. The Native API transaction model is based on ObjectScript transaction and locking methods, and is not interchangeable with the JDBC model. The Native API model must be used if your transactions include Native API method calls.

The following topics are discussed in this chapter:

- Controlling Transactions — describes how transactions are started, nested, rolled back, and committed.
- Concurrency Control — describes how to use the various lock methods.

Important: **Never Mix IRIS Native and JDBC Transaction Models**

NEVER mix the IRIS Native transaction model with the JDBC (java.sql) transaction model.

- If you want to use only IRIS Native commands within a transaction, you should always use IRIS Native transaction methods.
- If you want to use a mix of IRIS Native and JDBC/SQL commands within a transaction, you should turn autoCommit OFF and then always use IRIS 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 IRIS Native transaction methods.
- Although you can use both models in the same application, you must take care never to start a transaction in one model while a transaction is still running in the other model.

5.1 Controlling Transactions

The methods described here are alternatives to the standard JDBC transaction model. The Native API model for transaction and concurrency control is based on ObjectScript methods, and is not interchangeable with the JDBC model. The Native API model must be used if your transactions include Native API method calls.

For more information on the ObjectScript transaction model, see "Transaction Processing" in *Using ObjectScript*.

The Native API provides the following methods to control transactions:

- IRIS.**tCommit()** — commits one level of transaction.
- IRIS.**tStart()** — starts a transaction (which may be a nested transaction).
- IRIS.**getTLevel()** — returns an int value indicating the current transaction level (0 if not in a transaction).

- IRIS.**tRollback()** — rolls back all open transactions in the session.
- IRIS.**tRollbackOne()** — rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back.

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

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

```
    String globalName = "myGlobal";
    iris.tStart();

    // getTLevel() is 1: create myGlobal(1) = "firstValue"
    iris.set("firstValue", globalName, iris.getTLevel());

    iris.tStart();
    // getTLevel() is 2: create myGlobal(2) = "secondValue"
    iris.set("secondValue", globalName, iris.getTLevel());

    iris.tStart();
    // getTLevel() is 3: create myGlobal(3) = "thirdValue"
    iris.set("thirdValue", globalName, iris.getTLevel());

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

    iris.tRollbackOne();
    iris.tRollbackOne();  // roll back 2 levels to getTLevel 1
    iris.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 (iris.isDefined(globalName,ii) > 1) System.out.println(iris.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>
```

5.2 Concurrency Control

Concurrency control is a vital feature of multi-process systems such as InterSystems IRIS. It provides the ability to lock specific elements of data, preventing the corruption that would result from different processes changing the same element at the same time. The Native API transaction model provides a set of locking methods that correspond to ObjectScript commands. These methods must not be used with the JDBC/SQL transaction model (see the warning at the beginning of this chapter for details).

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

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

- 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.

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.

Tip: A detailed discussion of concurrency control is beyond the scope of this book. See the following books and articles for more information on this subject:

- "Transaction Processing" and "Lock Management" in *Using ObjectScript*
- "Locking and Concurrency Control" in the *Orientation Guide for Server-Side Programming*
- "LOCK" in the *ObjectScript Reference*

6 Native API for Java Quick Reference

This is a quick reference for the InterSystems IRIS Native API for Java, providing information on the following extension classes in com.intersystems.jdbc:

- Class IRIS provides the main functionality of the Native API.
- Class IRISIterator provides methods to navigate a global array.
- Class IRISList provides support for InterSystems $LIST serialization.
- Class IRISObject provides methods to work with Object Gateway reverse proxy objects.

All of these classes are part of the InterSystems JDBC driver (com.intersystems.jdbc). They access the database through a standard JDBC connection, and can be used without any special setup or installation procedures. For complete information on JDBC connections, see Establishing JDBC Connections in *Using Java with InterSystems Software*.

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

6.1 Class IRIS

Class IRIS is a member of com.intersystems.jdbc (the InterSystems JDBC driver).

IRIS has no public constructors. Instances of IRIS are created by calling static method IRIS.**createIRIS()**.

6.1.1 IRIS Method Details

classMethodBoolean()

jdbc.IRIS.**classMethodBoolean()** calls an ObjectScript class method, passing zero 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` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodBytes()

jdbc.IRIS.**classMethodBytes()** calls an ObjectScript class method, passing zero 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` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodDouble()

jdbc.IRIS.**classMethodDouble()** calls an ObjectScript class method, passing zero 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` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodIRISList()

jdbc.IRIS.**classMethodIRISList()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of IRISList.

```
final IRISList classMethodIRISList(String className, String methodName, Object... args)
```

This method is equivalent to `newList=(IRISList) classMethodBytes(className, methodName, args)`

parameters:

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodLong()

jdbc.IRIS.**classMethodLong()** calls an ObjectScript class method, passing zero 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` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodObject()

jdbc.IRIS.**classMethodObject()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of Object. If the returned object is a valid OREF (for example, if **%New()** was called), **classMethodObject()** will generate and return a reverse proxy object (an instance of IRISObject) for the referenced object. See "Using Java Reverse Proxy Objects" for details and examples.

```
final Object classMethodObject(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` — zero or more arguments of supported types.

classMethodStatusCode()

jdbc.IRIS.**classMethodStatusCode()** tests whether a class method that returns an ObjectScript $Status object would throw an error if called with the specified arguments. If the call would fail, this method throws a RuntimeException error containing the ObjectScript $Status error status number and message.

```
final void classMethodStatusCode(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` — zero or more arguments of supported types.

This is an indirect way to catch exceptions when using classMethod[type] calls. If the call would run without error, this method returns without doing anything, meaning that you can safely make the call with the specified arguments.

See "Calling ObjectScript Methods and Functions" for details and examples.

classMethodString()

jdbc.IRIS.**classMethodString()** calls an ObjectScript class method, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

classMethodVoid()

jdbc.IRIS.**classMethodVoid()** calls an ObjectScript class method with no return value, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

close()

jdbc.IRIS.**close()** closes the IRIS object.

```
final void close() throws Exception
```

createIRIS() [static]

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

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

parameters:

- `conn` — an instance of IRISConnection.

See “Introduction to Global Arrays” for more information and examples.

functionBoolean()

jdbc.IRIS.**functionBoolean()** calls an ObjectScript function, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

functionBytes()

jdbc.IRIS.**functionBytes()** calls an ObjectScript function, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

functionDouble()

jdbc.IRIS.**functionDouble()** calls an ObjectScript function, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

functionIRISList()

jdbc.IRIS.**functionIRISList()** calls an ObjectScript function, passing zero or more arguments and returning an instance of IRISList.

```
final IRISList functionIRISList(String functionName, String routineName, Object... args)
```

This function is equivalent to `newList=(IRISList) functionBytes(functionName, routineName, args)`

parameters:

- `functionName` — name of the function to call.
- `routineName` — name of the routine containing the function.
- `args` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

functionLong()

jdbc.IRIS.**functionLong()** calls an ObjectScript function, passing zero 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` — zero or more arguments of supported types.

See “Calling ObjectScript Methods and Functions” for details and examples.

functionObject()

jdbc.IRIS.**functionObject()** calls an ObjectScript function, passing zero or more arguments and returning an instance of Object. If the returned object is a valid OREF, **functionObject()** will generate and return a reverse proxy object (an instance of IRISObject) for the referenced object. See "Using Java Reverse Proxy Objects" for details and examples.

```
final Object functionObject(String functionName, String routineName, Object... args)
```

parameters:

- `functionName` — name of the function to call.
- `routineName` — name of the routine containing the function.
- `args` — zero or more arguments of supported types.

See "Class IRIS Supported Datatypes" for related information.

functionString()

jdbc.IRIS.**functionString()** calls an ObjectScript function, passing zero 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` — zero or more arguments of supported types.

See "Calling ObjectScript Methods and Functions" for details and examples.

getAPIVersion() [static]

jdbc.IRIS.**getAPIVersion()** returns the IRIS Native API version string.

```
static final String getAPIVersion()
```

getBoolean()

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

```
final Boolean getBoolean(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getBytes()

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

```
final byte[] getBytes(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getDate()

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

```
final java.sql.Date getDate(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getDouble()

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

```
final Double getDouble(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getFloat()

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

```
final Float getFloat(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getInputStream()

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

```
final InputStream getInputStream(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getInteger()

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

```
final Integer getInteger(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getIRISIterator()

jdbc.IRIS.**getIRISIterator()** returns an IRISIterator object (see “Class IRISIterator”) for the specified node. See “Iterating Over a Set of Child Nodes” for more information and examples.

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

parameters:

- `prefetchSizeHint` — (optional) hints number of bytes to fetch when getting data from server.
- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getIRISList()

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

```
IRISList getIRISList(String globalName, Object... subscripts)
```

This is equivalent to calling `newList=(IRISList) getBytes(globalName, subscripts)`

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getLong()

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

```
final Long getLong(String globalName, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getObject()

jdbc.IRIS.**getObject()** gets the value of the global as an Object (or `null` if node does not exist). See “Using Java Reverse Proxy Objects” for details and examples.

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

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

Use this method to retrieve objects that implement java.io.Serializable. When the **set()** method accepts an instance of Serializable, the instance is stored in serialized form, and can be deserialized by casting the return value of **getObject()** to the appropriate class.

See “Class IRIS Supported Datatypes” for related information.

getReader()

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

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

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

getServerVersion()

jdbc.IRIS.**getServerVersion()** returns the server version string for the current connection. This is equivalent to calling `$system.Version.GetVersion()` in ObjectScript.

```
final String getServerVersion()
```

getShort()

jdbc.IRIS.**getShort()** gets the value of the global as a Short (or `null` if node does not exist). Returns `0.0` if node value is an empty string.

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

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See “Class IRIS Supported Datatypes” for related information.

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, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

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, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getTimestamp()

jdbc.IRIS.**getTimestamp()** 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, Object... subscripts)
```

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

getTLevel()

jdbc.IRIS.**getTLevel()** gets the level of the current nested IRIS Native 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.

```
final Integer getTLevel()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

increment()

jdbc.IRIS.**increment()** increments the specified global with the passed value. If there is no node at the specified address, a new node is created with `value` as the value. A null value is interpreted as 0. Returns the new value of the global node. See "Creating, Accessing, and Deleting Nodes" for more information and examples.

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

parameters:

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

isDefined()

jdbc.IRIS.**isDefined()** returns a value indicating whether the specified node exists and if it contains a value. See "Finding Subnodes on All Levels" for more information and examples.

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

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

return values:

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

kill()

jdbc.IRIS.**kill()** deletes the global node including any descendants. See "Creating, Accessing, and Deleting Nodes" for more information and examples.

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

parameters:

- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

lock()

jdbc.IRIS.**lock()** locks the global for an IRIS Native transaction, and returns true on success. Note that this method performs an incremental lock and not the implicit unlock before lock feature that is also offered in ObjectScript.

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

parameters:

- `lockMode` — character `S` for shared lock, `E` for escalating lock, or `SE` for both. Default is empty string (exclusive and non-escalating).
- `timeout` — amount to wait to acquire the lock in seconds.
- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

procedure()

jdbc.IRIS.**procedure()** calls a procedure, passing zero or more arguments. Does not return a value.

```
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` — zero or more arguments of supported types.

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

releaseAllLocks()

jdbc.IRIS.**releaseAllLocks()** is an IRIS Native transaction method that releases all locks associated with the session.

```
final void releaseAllLocks()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

set()

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

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

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

parameters:

- `value` — value of a supported datatype (`null` value sets global to `""`).
- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

See "Class IRIS Supported Datatypes" for related information.

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 Java 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 — if the value is an instance of an object that implements Serializable, it will be serialized prior to being set as the global value. Use **getObject()** to retrieve the value.

tCommit()

jdbc.IRIS.**tCommit()** commits the current IRIS Native transaction.

```
final void tCommit()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

tRollback()

jdbc.IRIS.**tRollback()** rolls back all open IRIS Native transactions in the session.

```
final void tRollback()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

tRollbackOne()

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

```
final void tRollbackOne()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

tStart()

jdbc.IRIS.**tStart()** starts/opens an IRIS Native transaction.

```
final void tStart()
```

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

unlock()

jdbc.IRIS.**unlock()** unlocks the global in an IRIS Native transaction. 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, Object... subscripts)
```

parameters:

- `lockMode` — Character `S` for shared lock, `E` for escalating lock, or `SE` for both. Default is empty string (exclusive and non-escalating).
- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

This method uses the IRIS Native transaction model, and is not compatible with JDBC/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

6.2 Class IRISIterator

Class IRISIterator is a member of com.intersystems.jdbc (the InterSystems JDBC driver) and implements java.util.Iterator.

IRISIterator has no public constructors. Instances of IRISIterator are created by calling jdbc.IRIS.**getIRISIterator()**. See "Iterating Over a Set of Child Nodes" for more information and examples.

6.2.1 IRISIterator Method Details

getSubscriptValue()

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

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()

jdbc.IRISIterator.**getValue()** gets the value of the node at the current iterator position. 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()

jdbc.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()

jdbc.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()

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

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

previous()

jdbc.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()

jdbc.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()

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

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

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.

6.3 Class IRISList

Class IRISList is a member of com.intersystems.jdbc (the InterSystems JDBC driver). It implements a Java interface for InterSystems $LIST serialization. In addition to the IRISList constructors (described in the following section), it is also returned by the following jdbc.IRIS methods: **classMethodIRISList()**, **functionIRISList()**, **getIRISList()**.

6.3.1 IRISList Constructors

Constructor jdbc.IRISList.**IRISList()** has the following signatures:

```
IRISList ()
IRISList (IRISList list)
IRISList (byte[] buffer, int length)
```

parameters:

- `list` — instance of IRISList to be copied
- `buffer` — buffer to be allocated
- `length` — initial buffer size to be allocated

Instances of IRISList can be created in the following ways:

Create an empty IRISList

```
IRISList list = new IRISList();
```

Create a copy of another IRISList

Create a copy of the IRISList instance specified by argument *list*.

```
IRISList listcopy = new IRISList(myOtherList)
```

Construct an IRISList instance from a byte array

Construct an instance from a $LIST formatted byte array, such as that returned by IRIS.**getBytes()**. The constructor takes a *buffer* of size *length*:

```
byte[] listBuffer = myIris.getBytes("myGlobal",1);
IRISList listFromByte = new IRISList(listBuffer, listBuffer.length);
```

The returned list uses the buffer (not a copy) until changes to the list are visible in the buffer and a resize is required.

6.3.2 IRISList Method Details

add()

jdbc.IRISList.**add()** appends an Object, each element of an Object[], or each element of a Collection to the end of the IRISList. Throws RuntimeException if you attempt to add an unsupported datatype.

Supported datatypes are: Boolean, Short, Integer, Long, java.math.BigInteger, java.math.BigDecimal, Float, Double, String, Character, byte[],char[], IRISList, and `null`.

```
void add(Object value)
void add(Object[] array)
void add(Collection collection)
```

parameters:

- `value` — Object instance of any supported type.
- `array` — Object[] containing Object elements of supported types. Each Object in the array will be appended to the list as a separate element.
- `collection` — Collection containing Object elements of supported types. Each Object in the collection will be appended to the list in the order that it is returned by the collection's iterator.

Adding an IRISList element

The **add()** method never concatenates two instances of IRISList. Whenever **add()** encounters an IRISList instance (either as a single value or as an element of an array or collection), the instance is appended as a single IRISList element.

However, you can use IRISList.**toArray()** to convert an IRISList to an Object[]. Calling **add()** on the resulting array will append each element separately.

clear()

jdbc.IRISList.**clear()** resets the list by removing all elements from the list.

```
void clear()
```

count()

jdbc.IRISList.**count()** iterates over the list and returns the number of elements encountered.

```
int count()
```

DateToHorolog() [static]

jdbc.IRISList.**DateToHorolog()** converts a Date to the *day* field (int) of a **$Horolog** string. Also see **HorologToDate()**.

```
static int DateToHorolog(Date value, Calendar calendar)
```

parameters:

- `value` — Date value to convert.
- `calendar` — Calendar used to determine GMT and DST offsets. Can be null for default calendar

equals()

jdbc.IRISList.**equals()** compares the specified *list* with this instance of IRISList, and returns true if they are identical. To be equal, both lists must contain the same number of elements in the same order with identical serialized values.

```
boolean equals (Object list)
```

parameters:

- `list` — instance of IRISList to compare.

get()

jdbc.IRISList.**get()** returns the element at *index* as Object. Throws IndexOutOfBoundsException if the index is less than 1 or past the end of the list.

```
Object get(int index)
```

parameters:

- `index` — integer specifying the list element to be retrieved.

getList()

jdbc.IRISList.**getList()** gets the element at *index* as an IRISList. Throws IndexOutOfBoundsException if the index is less then 1 or past the end of the list.

```
IRISList getList(int index)
```

parameters:

- `index` — integer specifying the list element to return

HorologToDate() [static]

jdbc.IRISList.**HorologToDate()** converts the *day* field of a **$Horolog** string to a Date value. Also see **DateToHorolog()**.

```
static Date HorologToDate(int HorologValue, Calendar calendar)
```

parameters:

- `HorologValue` — int representing the *day* field of a **$Horolog** string. The time field will be zeroed out (i.e., midnight), so a Date will not round trip if its time fields are not zero to begin with.
- `calendar` — Calendar used to determine GMT and DST offsets. Can be null for default calendar.

HorologToTime() [static]

jdbc.IRISList.**HorologToTime()** converts the *time* field of a **$Horolog** string to a Time value. Also see **TimeToHorolog()**.

```
static Time HorologToTime(int HorologValue, Calendar calendar)
```

parameters:

- `HorologValue` — int representing the *time* field of a **$Horolog** string. The *date* field will be zeroed out (set to the epoch), so a Time with a milliseconds value outside the range `0L` to `86399999L` does not round trip.
- `calendar` — Calendar used to determine GMT and DST offsets. Can be null for default calendar

PosixToTimestamp() [static]

jdbc.IRISList.**PosixToTimestamp()** converts a %Library.PosixTime value to a Timestamp. Treats the %PosixTime as local time using the supplied Calendar (if null, uses the default time zone). To round trip, you must supply equivalent calendars. Also see **TimestampToPosix()**.

```
static Timestamp PosixToTimestamp(long PosixValue, Calendar calendar)
```

parameters:

- `PosixValue` — the %PosixTime value (a 64-bit signed integer).
- `calendar` — Calendar, used to determine GMT and DST offsets. Can be null for default calendar

remove()

jdbc.IRISList.**remove()** removes the element at *index* from the list. Returns true if the element existed and was removed, false otherwise. This method can be expensive, since it will usually have to reallocate the list.

```
boolean remove(int index)
```

parameters:

- `index` — integer specifying the list element to remove.

set()

jdbc.IRISList.**set()** sets or replaces the list element at *index* with *value*. If *value* is an array, each array element is inserted into the list, starting at *index*, and any existing list elements after *index* are shifted to make room for the new values. If *index* is beyond the end of the list, *value* will be stored at *index* and the list will be padded with nulls up to that position.

```
void set(int index, Object value)
void set(int index, Object[] value)
```

parameters:

- `index` — integer indicating the list element to be set or replaced
- `value` — Object value or Object array to insert at *index*

Objects can be any of the following types: Boolean, Short, Integer, Long, java.math.BigInteger, java.math.BigDecimal, Float, Double, String, Character, byte[],char[], IRISList, and `null`.

size()

jdbc.IRISList.**size()** returns the byte length of the serialized value for this IRISList.

```
int size()
```

subList()

jdbc.IRISList.**subList()** returns a new IRISList containing the elements in the closed range [*from*, *to*]. Throws IndexOutOfBoundsException if *from* is greater than **count()** or *to* is less than *from*.

```
IRISList subList(int from, int to)
```

parameters:

- `from` — index of first element to add to the new list.
- `to` — index of last element to add to the new list.

TimestampToPosix() [static]

jdbc.IRISList.**TimestampToPosix()** converts a Timestamp object to a %Library.PosixTime value (a 64-bit signed integer). Converts the Timestamp to local time using the supplied Calendar (if null, uses the default time zone). To round trip, you must supply equivalent calendars. Also see **PosixToTimestamp()**.

```
static long TimestampToPosix(Timestamp Value, Calendar calendar)
```

parameters:

- `Value` — Timestamp to be converted.
- `calendar` — Calendar used to determine GMT and DST offsets. Can be null for default calendar

TimeToHorolog() [static]

jdbc.IRISList.**TimeToHorolog()** converts a Time value to the *time* field of a **$Horolog** string. Also see **HorologToTime()**.

```
static int TimeToHorolog(Time value, Calendar calendar)
```

parameters:

- `value` — Time to be converted.
- `calendar` — Calendar used to determine GMT and DST offsets. Can be null for default calendar

toArray()

jdbc.IRISList.**toArray()** returns an array of Object containing all of the elements in this list. If the list is empty, a zero length array will be returned.

```
Object[] toArray()
```

toList()

jdbc.IRISList.**toList()** returns a java.util.ArrayList<Object> containing all of the elements in this IRISList. If the list is empty, a zero length ArrayList will be returned.

```
ArrayList< Object> toList()
```

toString()

jdbc.IRISList.**toString()** returns a printable representation of the list.

```
String toString()
```

Some element types are represented differently than they would be in ObjectScript:

- An empty list (`""` in ObjectScript) is displayed as `"$lb()"`.
- Empty elements (where `$lb()=$c(1)`) are displayed as `"null"`.
- Strings are not quoted.
- Doubles format with sixteen significant digits

6.4 Class IRISObject

Class IRISObject is a member of com.intersystems.jdbc (the InterSystems JDBC driver). It provides methods to work with Object Gateway reverse proxy objects (see "Using Java Reverse Proxy Objects" for details and examples).

IRISObject has no public constructors. Instances of IRISObject can be created by calling one of the following IRIS methods:

- jdbc.IRIS.**classMethodObject()**
- jdbc.IRIS.**functionObject()**

If the called method or function returns an object that is a valid OREF, a reverse proxy object (an instance of IRISObject) for the referenced object will be generated and returned. For example, **classMethodObject()** will return a proxy object for an object created by **%New()**.

See "Using Java Reverse Proxy Objects" for details and examples.

6.4.1 IRISObject Method Details

close()

jdbc.IRISObject.**close()** closes the object.

```
void close()
```

get()

jdbc.IRISObject.**get()** returns a property value of the proxy object as an instance of Object.

```
Object get(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

getBoolean()

jdbc.IRISObject.**getBoolean()** returns a property value of the proxy object as an instance of Boolean.

```
Boolean getBoolean(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See "IRISObject Supported Datatypes" for related information.

getBytes()

jdbc.IRISObject.**getBytes()** returns a property value of the proxy object as an instance of byte[].

```
byte[] getBytes(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See "IRISObject Supported Datatypes" for related information.

getDouble()

jdbc.IRISObject.**getDouble()** returns a property value of the proxy object as an instance of Double.

```
Double getDouble(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See “IRISObject Supported Datatypes” for related information.

getIRISList()

jdbc.IRISObject.**getIRISList()** returns a property value of the proxy object as an instance of IRISList.

```
IRISList getIRISList(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See “IRISObject Supported Datatypes” for related information.

getLong()

jdbc.IRISObject.**getLong()** returns a property value of the proxy object as an instance of Long.

```
Long getLong(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See “IRISObject Supported Datatypes” for related information.

getObject()

jdbc.IRISObject.**getObject()** returns a property value of the proxy object as an instance of Object.

```
Object getObject(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See “IRISObject Supported Datatypes” for related information.

getString()

jdbc.IRISObject.**getString()** returns a property value of the proxy object as an instance of String.

```
String getString(String propertyName)
```

parameters:

- `propertyName` — name of the property to be returned.

See “IRISObject Supported Datatypes” for related information.

invoke()

jdbc.IRISObject.**invoke()** invokes an instance method of the object, returning value as Object.

```
Object invoke(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See "IRISObject Supported Datatypes" for related information.

invokeBoolean()

jdbc.IRISObject.**invokeBoolean()** invokes an instance method of the object, returning value as Boolean.

```
Boolean invokeBoolean(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See "IRISObject Supported Datatypes" for related information.

invokeBytes()

jdbc.IRISObject.**invokeBytes()** invokes an instance method of the object, returning value as byte[].

```
byte[] invokeBytes(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See "IRISObject Supported Datatypes" for related information.

invokeDouble()

jdbc.IRISObject.**invokeDouble()** invokes an instance method of the object, returning value as Double.

```
Double invokeDouble(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See "IRISObject Supported Datatypes" for related information.

invokeIRISList()

jdbc.IRISObject.**invokeIRISList()** invokes an instance method of the object, returning value as IRISList.

```
IRISList invokeIRISList(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See “IRISObject Supported Datatypes” for related information.

invokeLong()

jdbc.IRISObject.**invokeLong()** invokes an instance method of the object, returning value as Long.

```
Long invokeLong(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See “IRISObject Supported Datatypes” for related information.

invokeObject()

jdbc.IRISObject.**invokeObject()** invokes an instance method of the object, returning value as Object

```
Object invokeObject(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See “IRISObject Supported Datatypes” for related information.

invokeStatusCode()

jdbc.IRISObject.**invokeStatusCode()** tests whether an invoke method that returns an ObjectScript $Status object would throw an error if called with the specified arguments. If the call would fail, this method throws a RuntimeException error containing the ObjectScript $Status error number and message. See “Catching %Status Error Codes” for details and examples.

```
void invokeStatusCode(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See “IRISObject Supported Datatypes” for related information.

invokeString()

jdbc.IRISObject.**invokeString()** invokes an instance method of the object, returning value as String.

```
String invokeString(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See “IRISObject Supported Datatypes” for related information.

invokeVoid()

jdbc.IRISObject.**invokeVoid()** invokes an instance method of the object, but does not return a value.

```
void invokeVoid(String methodName, Object... args)
```

parameters:

- `methodName` — name of the instance method to be called.
- `args` — zero or more arguments of supported types.

See "IRISObject Supported Datatypes" for related information.

iris [attribute]

jdbc.IRISObject.**iris** is a public field that provides access to the instance of IRIS associated with this object.

```
public IRIS iris
```

See "IRISObject Supported Datatypes" for related information.

set()

jdbc.IRISObject.**set()** sets a property of the proxy object.

```
void set(String propertyName, Object propertyValue)
```

parameters:

- `propertyName` — name of the property to which *value* will be assigned.
- `value` — property value to assign.