InterSystems™
IRIS Data Platform

Using the Native API for Java

Version 2019.4
2020-01-28

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

Table of Contents

List of Figures

About This Book

The InterSystems *Native API for Java* is a lightweight interface to powerful InterSystems IRIS® resources that were once available only through ObjectScript:

- *Call ObjectScript classmethods and functions*. Write custom ObjectScript classmethods or functions for any purpose, and call them from your Java application as easily as you can call native Java methods.
- *Manipulate ObjectScript class instances* through Object Gateway proxy objects. Call instance methods and get or set property values as easily as you could in an ObjectScript application.
- *Directly access globals*, the tree-based sparse arrays used to implement the InterSystems multidimensional storage model. These native data structures provide very fast, flexible storage and retrieval. InterSystems IRIS uses globals to make data available as objects or relational tables, but you can use the Native API to implement your own data structures.

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

- Introduction to the Native API — gives an overview of Native API abilities and provides some simple code examples.
- Calling ObjectScript Methods and Functions — describes how to call user defined ObjectScript class methods 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. With the Native API, your applications can take advantage of seamless InterSystems IRIS data platform integration:

- Implement transparent bidirectional communication between ObjectScript and Java

  The Native API, Object Gateway proxy objects, and ObjectScript applications can all share the same connection and work together in the same context.

- Create and use individual instances of an ObjectScript class

  The Native API allows you to create instances of ObjectScript classes on InterSystems IRIS and generate Object Gateway proxy objects for them at runtime. Your Java application can use proxies to work with ObjectScript objects as easily as if they were native Java objects.

- Call ObjectScript classmethods and user-defined functions

  You can write custom ObjectScript classmethods or functions for any purpose, and your application can use the Native API to call them just as easily as native Java methods.

- Work with multidimensional global arrays

  The Native API provides direct access to the high performance native data structures (global arrays) that underpin the InterSystems IRIS multidimensional storage model. 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 all of these abilities to your Java application.

Implement transparent bidirectional communication between ObjectScript and Java

The Native API for Java is implemented as an extension to the InterSystems JDBC driver in class jdbc.IRIS. Connections are created just they would be for any other Java application (see *Using Java with InterSystems Software*). This example opens a connection and then creates an instance of the Native API:

```
//Open a connection to InterSystems IRIS
  String connStr = "jdbc:IRIS://127.0.0.1:51773/USER";
  IRISConnection conn = (IRISConnection) DriverManager.getConnection(connStr,user,pwd);

// Use the connection to create an instance of the Native API
  IRIS iris = IRIS.createIRIS(conn);
```

This connection can also be used by the InterSystems Object Gateway, allowing your Java and ObjectScript applications to share the same context and work with the same objects.

Create and use individual instances of an ObjectScript class

Your application can create an instance of an ObjectScript class, immediately generate an Object Gateway proxy for it, and use the proxy to work with the ObjectScript instance (see the chapter on "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.

Call ObjectScript classmethods and user-defined functions

You can easily call an ObjectScript classmethod or function (see the chapter on "Calling ObjectScript Methods and Functions").

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

This example just calls a classmethod to get some system information, but the real power of these calls is their ability to leverage user-written code. You can write custom ObjectScript classmethods or functions for any purpose, and your Java application can call them as easily as it calls native Java methods.

Work with multidimensional global arrays

The Native API provides all the methods needed to manipulate global arrays (see the chapter on "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 that call class methods and functions from ObjectScript classes and routines. They allow Java applications to call your custom ObjectScript classmethods or functions as easily as they call native Java methods. See the following sections for details and examples:

- Class Method Calls — demonstrates how to call user defined ObjectScript class methods.
- Function Calls — demonstrates how to call user defined ObjectScript functions and procedures.

Note: In some cases, these Native API methods can also be used with InterSystems classes defined in the Class Library. For example, this call returns some system information:

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

Unfortunately, most class library methods return only a status code, passing the actual results back in arguments (which cannot be accessed by the Native API). Best practice is always to call class library methods indirectly, from within a user defined wrapper class. This allows you to use the full power of the class library, and also produces more maintainable code.

2.1 Class Method Calls

The Native API methods in this section call user defined ObjectScript class methods, and return values of the type indicated by the method name: **classMethodBoolean()**, **classMethodBytes()**, **classMethodDouble()**, **classMethodLong()**, **classMethodString()**, or **classMethodVoid()** (no return value).

They take String arguments for *className* and *methodName*, plus `0` or more method arguments. 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.

In addition to these methods, you can use **classMethodStatusCode()** to retrieve error information when an ObjectScript class method call fails. When called from within a catch clause (as demonstrated at the end of the example below) it will throw a RuntimeException error containing the ObjectScript error status number and message.

Calling ObjectScript class methods with the Native API

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 *iris* is an existing instance of class IRIS, and is currently connected to the server.

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

  System.out.print("cmBoolean() finds if two numbers are equal (true=1,false=0): ");
  System.out.println(iris.classMethodBoolean(className,"cmBoolean",7,7));

  System.out.print("cmBytes() converts a list of integers into a byte array: ");
  System.out.println(new String(iris.classMethodBytes(className,"cmBytes",72,105,33)));

  System.out.print("cmString() concatenates two strings: ");
  System.out.println(iris.classMethodString(className,"cmString","World"));

  System.out.print("cmLong() returns the total of two numbers: ");
  System.out.println(iris.classMethodLong(className,"cmLong",7,8));

  System.out.print("cmDouble() multiplies a number by 100: ");
  System.out.println(iris.classMethodDouble(className,"cmDouble",7.567));

  System.out.print("cmVoid assigns a value to a global array: ");
  iris.classMethodVoid(className,"cmVoid",67);
  System.out.println("^cmGlobal = "_iris.getInteger("^cmGlobal"));

// Get an error message for a failed method call
  try {
    System.out.println("Calling nonexistent class method \"notLegal\"... ");
    iris.classMethodStatusCode(className,"notLegal");
  } catch (RuntimeException e) {
    System.out.println("Call to class method \"notLegal\" returned error:");
    System.out.println(e.getMessage());
  }
```

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 * 100
  }
  ClassMethod cmVoid(cm1 As %Integer)
  {
   set ^cmGlobal=cm1
   quit
  }
}
```

2.2 Function Calls

The Native API methods in this section call user-defined ObjectScript functions or procedures (see “Callable User-defined Code Modules” in *Using ObjectScript*), and return a value of the type indicated by the method name: **functionBoolean()**, **functionBytes()**, **functionDouble()**, **functionLong()**, **functionString()**, or **procedure()** (no return value).

They take String arguments for *functionName* and *routineName*, plus `0` or more function arguments. 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: These methods are designed to call functions in user-defined routines. ObjectScript system functions (see “ObjectScript Functions” in the *ObjectScript Reference*) cannot be called directly from your Java code. However, you can write an ObjectScript wrapper function to call a system function indirectly. For instance, the sample code later in this section includes user-defined function **functionBytes()**, which returns the value of a call to ObjectScript system function $CHAR.

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";

System.out.print("funcBoolean() finds if two numbers are equal (true=1,false=0): ");
System.out.println(iris.functionBool("funcBoolean",routineName,7,7));

System.out.print("funcBytes() converts a list of integers into a byte array: ");
System.out.println(new String(iris.functionBytes("funcBytes",routineName,72,105,33)));

System.out.print("funcString() concatenates two strings: ");
System.out.println(iris.functionString("funcString",routineName,"World"));

System.out.print("funcLong() returns the total of two numbers: ");
System.out.println(iris.functionInt("funcLong",routineName,7,8));

System.out.print("funcDouble() multiplies a number by 100: ");
System.out.println(iris.functionDouble("funcDouble",routineName,7.567));

System.out.print("funcProcedure assigns a value to a global array: ");
iris.procedure("funcProcedure",routineName,88);
System.out.println("^funcGlobal = "_iris.getInteger("^funcGlobal"));
```

ObjectScript Routine NativeRoutine.mac

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

```
funcBoolean(fn1,fn2) public {
   quit (fn1=fn2)
}
funcBytes(fn1,fn2,fn3) public {
    quit $CHAR(fn1,fn2,fn3)
}
funcString(fn1) public {
    quit "Hello "_fn1
}
funcLong(fn1,fn2) public {
    quit fn1+fn2
}
funcDouble(fn1) public {
    quit fn1 * 100
}
funcProcedure(fn1) public {
    set ^funcGlobal=fn1
    quit
}
```

3

Using Java Reverse Proxy Objects

The Native API works together with the Object Gateway. *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.

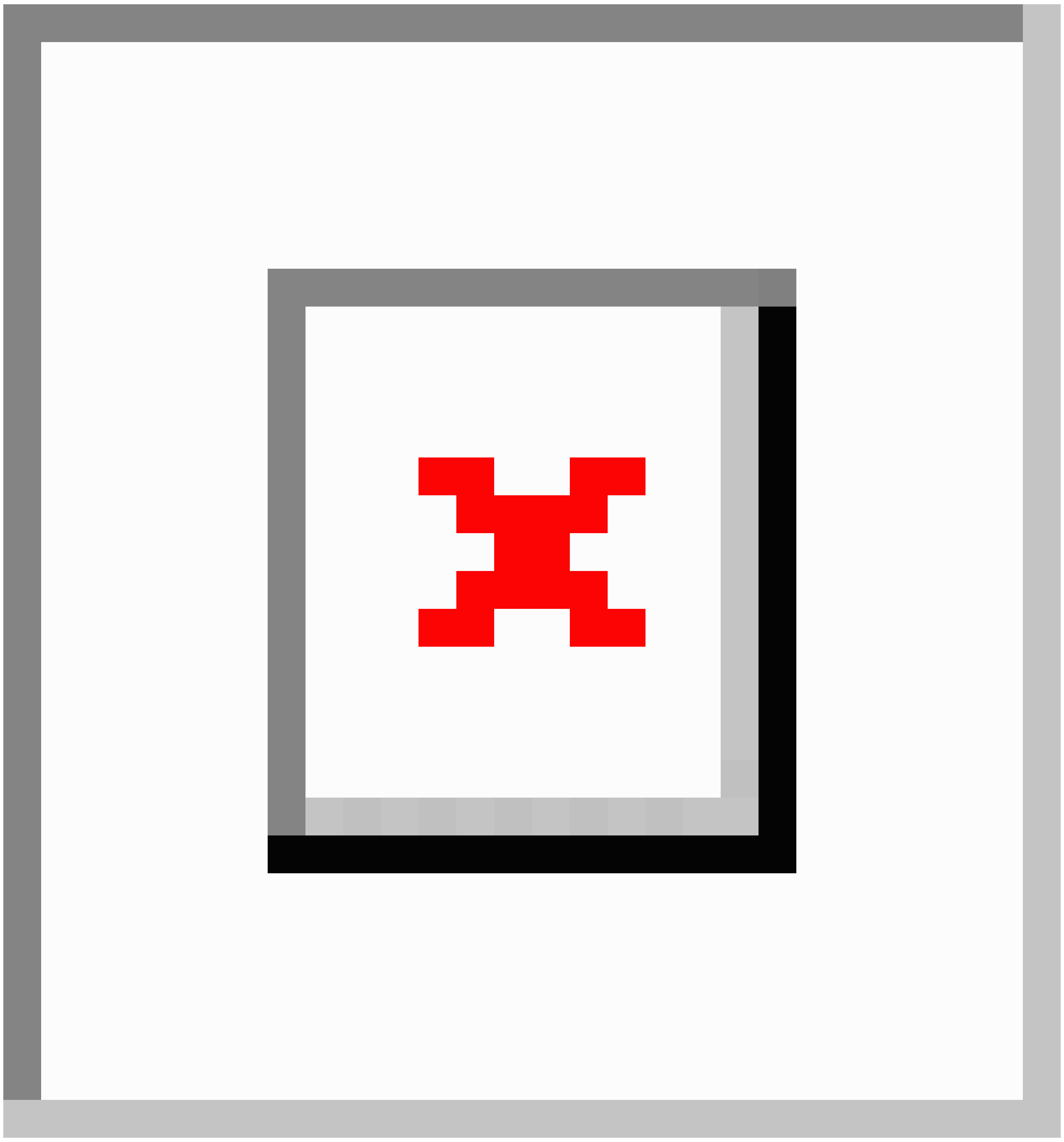
This chapter covers the following topics:

- 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 Methods for Working with Reverse Proxies — describes some jdbc.IRIS methods that provide an alternate interface for reverse proxy objects.

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*, 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 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 four methods to control the ObjectScript target object: **invoke()** and **invokeVoid()** call an instance method with or without a return value, and accessors **get()** and **set()** get and set a property value.

Assume that we have created an ObjectScript class named Demo.Test, which includes declarations for methods **initialize()** and **add()**, and property *name* (we won't need the actual method code for this example):

```
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 the proxy to access the target object.

Using reverse proxy object methods

```
// 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
```

In this example, IRISObject proxy methods are used to access methods and properties of the Demo.Test instance:

- **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*.

The IRIS.**classMethodObject()** call was discussed previously in "Creating Reverse Proxy Objects".

3.4 IRIS Methods for Working with Reverse Proxies

The IRIS class includes four methods that allow you to manipulate IRISObject instances by reference. Each method corresponds to one of the IRISObject methods, as shown below:

jdbc.IRISObject Methods	jdbc.IRIS Methods
invoke()	**instanceMethodObject()**
`test.invoke("add", 2, 2);`	`iris.instanceMethodObject(test,"add",2,2);`
invokeVoid()	**instanceMethodVoid()**
`test.invoke("init", 0);`	`iris.instanceMethodObject(test,"init",0);`
set()	**setInstanceProperty()**
`test.set("name", "Albert");`	`iris.setInstanceProperty(test,"name", "Albert");`
get()	**getInstanceProperty()**
`String name = test.get("name");`	`String name = iris.getInstanceProperty(test,"name");`

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, Updating, and Deleting Nodes — demonstrates how to create, change, or delete nodes in a global array.
- Finding Nodes in a Global Array — describes the iteration methods that allow rapid access to the nodes of a global array.

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, Updating, 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

The Native API for Java is an extension to the InterSystems JDBC driver. For detailed information on installation and usage, see the "Configuration and Requirements" in *Using Java with InterSystems Software*.

```
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://127.0.0.1:51773/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, Updating, and Deleting Nodes

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

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

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

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

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

Creating and Deleting Nodes: Setting and incrementing node values

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

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

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

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

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

Deleting Nodes with kill()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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 Transaction Models**

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

- Always use the Native API transaction model if your transaction will include any of the Native API methods described in this book.
- Native API transactions can also include any JDBC/SQL methods other than transaction methods.
- If you use only JDBC/SQL commands within a transaction, you can use either model.
- 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.

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 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 IRISObject provides methods to work with Object Gateway reverse proxy objects.

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.

6.1 Class IRIS

Class IRIS is a member of com.intersystems.jdbc.

Instances of IRIS are created by calling static method IRIS.**createIRIS()**.

6.1.1 Method Details

classMethodBoolean()

jdbc.IRIS.**classMethodBoolean()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of Boolean. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

classMethodBytes()

jdbc.IRIS.**classMethodBytes()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of byte[]. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

classMethodDouble()

jdbc.IRIS.**classMethodDouble()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of Double. See “Calling ObjectScript Methods and Functions” for details and examples.

```
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.

classMethodLong()

jdbc.IRIS.**classMethodLong()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of Long. See “Calling ObjectScript Methods and Functions” for details and examples.

```
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.

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 an ObjectScript class method would throw an error if called with the specified arguments. See “Calling ObjectScript Methods and Functions” for details and examples.

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. If call would fail, it throws a RuntimeException error containing the ObjectScript 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.

classMethodString()

jdbc.IRIS.**classMethodString()** calls an ObjectScript class method, passing zero or more arguments and returning an instance of String. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

classMethodVoid()

jdbc.IRIS.**classMethodVoid()** calls an ObjectScript class method with no return value, passing zero or more arguments. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

close()

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

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

createIRIS()

jdbc.IRIS.**createIRIS()** returns an instance of jdbc.IRIS that uses the specified IRISConnection. See "Introduction to Global Arrays" for more information and examples.

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

parameters:

- `conn` — an instance of IRISConnection.

functionBoolean()

jdbc.IRIS.**functionBoolean()** calls an ObjectScript function, passing zero or more arguments and returning an instance of Boolean. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

functionBytes()

jdbc.IRIS.**functionBytes()** calls an ObjectScript function, passing zero or more arguments and returning an instance of byte[]. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

functionDouble()

jdbc.IRIS.**functionDouble()** calls an ObjectScript function, passing zero or more arguments and returning an instance of Double. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

functionLong()

jdbc.IRIS.**functionLong()** calls an ObjectScript function, passing zero or more arguments and returning an instance of Long. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

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.

functionString()

jdbc.IRIS.**functionString()** calls an ObjectScript function, passing zero or more arguments and returning an instance of String. See "Calling ObjectScript Methods and Functions" for details and examples.

```
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.

getAPIVersion()

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.

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.

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.

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.

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.

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.

getInstanceProperty()

jdbc.IRIS.**getInstanceProperty()** gets the value of the specified instance property as an Object (also see IRISObject method **get()**). See "Using Java Reverse Proxy Objects" for details and examples.

```
public final Object getInstanceProperty(IRISObject instance, String propertyName)
```

parameters:

- `instance` — the IRISObject instance to be accessed.
- `propertyName` — name of the instance property.

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.

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.

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.

getObject()

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

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

parameters:

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

getReader()

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

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

parameters:

- `globalName` — global name.

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

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, String... subscripts )
```

parameters:

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

getString()

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

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

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

parameters:

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

getTime()

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

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

parameters:

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

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.

getTLevel()

jdbc.IRIS.**getTLevel()** gets the level of the current nested transaction. Returns 1 if there is only a single transaction open. Returns 0 if there are no transactions open. This is equivalent to fetching the value of the **$TLEVEL** special variable. See "Transactions and Locking" for more information and examples.

```
final Integer getTLevel ( )
```

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, Updating, and Deleting Nodes" for more information.

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

parameters:

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

instanceMethodObject()

jdbc.IRIS.**instanceMethodObject()** invokes an instance method of the proxy object, returning the value as Object. See "Using Java Reverse Proxy Objects" for details and examples.

```
final Object instanceMethodObject(IRISObject instance, String methodName, Object... args)
```

parameters:

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

instanceMethodVoid()

jdbc.IRIS.**instanceMethodVoid()** invokes an instance method of the proxy object, but does not return a value. See "Using Java Reverse Proxy Objects" for details and examples.

```
final void instanceMethodVoid(IRISObject instance, String methodName, Object... args)
```

parameters:

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

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, String... 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, Updating, and Deleting Nodes" for more information and examples.

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

parameters:

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

lock()

jdbc.IRIS.**lock()** locks the global, returns true on success. Note that this method performs an incremental lock and not the implicit unlock before lock feature that is also offered in ObjectScript. See "Transactions and Locking" for more information and examples.

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

parameters:

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

procedure()

jdbc.IRIS.**procedure()** calls a procedure, passing zero or more arguments. Does not return a value. See "Calling ObjectScript Methods and Functions" for more information and examples.

```
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.

releaseAllLocks()

jdbc.IRIS.**releaseAllLocks()** releases all locks associated with the session. See "Transactions and Locking" for more information and examples.

```
final void releaseAllLocks ( )
```

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, Updating, and Deleting Nodes" for more information.

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

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

parameters:

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

Notes on specific datatypes

The following datatypes have some extra features:

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

setInstanceProperty()

jdbc.IRIS.**setInstanceProperty()** sets a property of the proxy object. Also see IRISObject method **set()**. See "Using Java Reverse Proxy Objects" for details and examples.

```
final void setInstanceProperty(IRISObject instance, String propertyName, Object value)
```

parameters:

- `instance` — the IRISObject instance to be accessed.
- `propertyName` — name of the property.
- `value` — the property value to be assigned.

tCommit()

jdbc.IRIS.**tCommit()** commits the current transaction. See “Transactions and Locking” for more information and examples.

```
final void tCommit ( )
```

tRollback()

jdbc.IRIS.**tRollback()** rolls back all open transactions in the session. See “Transactions and Locking” for more information and examples.

```
final void tRollback ( )
```

tRollbackOne()

jdbc.IRIS.**tRollbackOne()** rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back. See “Transactions and Locking” for more information and examples.

```
final void tRollbackOne ( )
```

tStart()

jdbc.IRIS.**tStart()** starts/opens a transaction. See “Transactions and Locking” for more information and examples.

```
final void tStart ( )
```

unlock()

jdbc.IRIS.**unlock()** unlocks the global. This method performs an incremental unlock, not the implicit unlock-before-lock feature that is also offered in ObjectScript. See “Transactions and Locking” for more information and examples.

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

parameters:

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

6.2 Class IRISIterator

Class IRISIterator is a member of com.intersystems.jdbc.

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 Method and Property 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 IRISObject

Class IRISObject is a member of com.intersystems.jdbc.

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.

6.3.1 IRISObject Method Details

close()

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

```
void close()
```

get()

jdbc.IRISObject.**get()** returns a property of the object as Object (also see jdbc.IRIS.**getInstanceProperty()**).

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

parameters:

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

invoke()

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

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

parameters:

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

invokeVoid()

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

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

parameters:

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

set()

jdbc.IRISObject.**set()** sets a property of the object (also see jdbc.IRIS.**setInstanceProperty()**).

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

parameters:

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