Using the Native SDK for .NET

Version 2024.1
2024-05-02

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

Using the Native SDK for .NET
InterSystems IRIS Data Platform Version 2024.1 2024-05-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1 Native SDK for .NET Overview 1

2 Introduction to the .NET Native SDK 3

3 Calling ObjectScript Methods and Functions from .NET 5
- 3.1 Class Method Calls 5
- 3.2 Function Calls 7
- 3.3 Calling Class Library Methods 8
  - 3.3.1 Using Pass-by-reference Arguments 8
  - 3.3.2 Catching %Status Error Codes 9

4 Using .NET Inverse Proxy Objects 11
- 4.1 Introducing External Servers 11
- 4.2 Creating an Inverse Proxy Object 12
- 4.3 Controlling a Target Object 12
- 4.4 IRISObject Supported Datatypes 13

5 Working with Global Arrays 15
- 5.1 Introduction to Global Arrays 15
  - 5.1.1 Glossary of Native SDK Terms 17
  - 5.1.2 Global Naming Rules 18
- 5.2 Creating, Accessing, and Deleting Nodes 19
  - 5.2.1 Creating Nodes and Setting Node Values 19
  - 5.2.2 Getting Node Values 20
  - 5.2.3 Deleting Nodes 20
- 5.3 Finding Nodes in a Global Array 20
  - 5.3.1 Iterating Over a Set of Child Nodes 21
  - 5.3.2 Iteration in Conditional Loops 22
  - 5.3.3 Testing for Child Nodes and Node Values 23
- 5.4 Class IRIS Supported Datatypes 24

6 Transactions and Locking 27
- 6.1 Controlling Transactions 27
- 6.2 Concurrency Control 28

7 Native SDK for .NET Quick Reference 31
- 7.1 Class IRIS 31
  - 7.1.1 IRIS Method Details 31
- 7.2 Class IRISIterator 43
  - 7.2.1 IRISIterator Method and Property Details 44
- 7.3 Class IRISList 44
  - 7.3.1 IRISList Constructors 45
  - 7.3.2 IRISList Method Details 45
- 7.4 Class IRISObject 49
  - 7.4.1 IRISObject Method Details 49

List of Figures

Figure 4–1: External Server connection 11

1

Native SDK for .NET Overview

See the Table of Contents for a detailed listing of the subjects covered in this document.

The InterSystems *Native SDK for .NET* is a lightweight interface to powerful InterSystems IRIS® resources that were once available only in ObjectScript:

- *Call ObjectScript classmethods and functions*. Write custom ObjectScript classmethods or functions for any purpose, and call them from your .NET application as easily as you can call native .NET methods.
- *Manipulate ObjectScript class instances* through 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 SDK to implement your own data structures.

The following chapters discuss the main features of the Native SDK for .NET:

- Introduction to the .NET Native SDK — gives an overview of Native SDK abilities and provides some simple code examples.
- Calling ObjectScript Methods and Functions from .NET — describes how to call ObjectScript classmethods and functions from .NET applications.
- Using .NET Inverse Proxy Objects — demonstrates how to use .NET inverse proxy objects to control ObjectScript objects.
- Working with Global Arrays in .NET — 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 .NET Native SDK transaction and concurrency control model.
- .NET Native SDK Quick Reference — provides a brief description of each .NET Native SDK method mentioned in this book.

InterSystems Core SDKs for Java

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

- *Using .NET with InterSystems Software* — describes how use the InterSystems implementation of the ADO.NET Managed Provider for relational data access and the Entity Framework for object access.

- *Persisting .NET Objects with XEP* — describes how to store and retrieve .NET objects using the InterSystems XEP SDK.

Native SDKs for other languages

Versions of the Native SDK are also available for Java, Python, and Node.js:

- *Using the Native SDK for Java*
- *Using the Native SDK for Python*
- *Using the Native SDK for Node.js*

More information about globals

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.

2

Introduction to the .NET Native SDK

The *Native SDK for .NET* is a lightweight interface to powerful InterSystems IRIS® resources that were once available only through ObjectScript. With the Native SDK, your applications can take advantage of seamless InterSystems IRIS data platform integration:

- Implement transparent bidirectional communication between ObjectScript and .NET

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

- Work with multidimensional global arrays

  The Native SDK 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 .NET applications just as they can in ObjectScript.

Important: To use the Native SDK for .NET, you must download the .NET connection package as described in Connection Tools.

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

Implement transparent bidirectional communication between ObjectScript and .NET

The Native SDK for .NET is implemented as an extension to the InterSystems ADO.NET Managed Provider. Connections are created just they would be for any other application using the Managed Provider (see *Using the InterSystems Managed Provider for .NET*). This example opens a connection and then creates an instance of the Native SDK IRIS class:

```
//Open a connection to InterSystems IRIS
  IRISConnection conn = new IRISConnection();
  conn.ConnectionString = "Server = localhost; " + "Port = 1972; "
    + "Namespace = USER; " + "Password = SYS; " + "User ID = _SYSTEM;";
  conn.Open();

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

This connection can also be used by the InterSystems Object Gateway, allowing your .NET 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 .NET Inverse Proxy Objects").

In this example, the first line calls the **%New()** method of ObjectScript class Demo.dataStore, creating an instance in InterSystems IRIS. In .NET, 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 .NET 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 .NET application can call them as easily as it calls native .NET methods.

Work with multidimensional global arrays

The Native SDK 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
  Console.Write("New number is " + iris.GetInteger("myGlobal", "subOne"));
  iris.Kill("myGlobal", "subOne");
```

3

Calling ObjectScript Methods and Functions from .NET

This chapter describes methods of class IRIS that allow you to call ObjectScript class methods and user-defined functions directly from your .NET 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.

3.1 Class Method Calls

The following methods of class IRIS call ObjectScript class methods, returning values of the type indicated by the method name: **ClassMethodBool()**, **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: int?, short?, string, long?, double?, float?, byte[], bool?, DateTime?, IRISList?, or IRISObject. If the connection is bidirectional (see "Using .NET Inverse Proxy Objects"), then any .NET object can be used as an argument. See "Class IRIS Supported Datatypes" for more information about how the Native SDK handles these datatypes.

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 = "";
try{
comment = "cmBoolean() tests whether two numbers are equal (true=1,false=0): ";
bool boolVal = iris.ClassMethodBool(className,"cmBoolean",7,7);
Console.WriteLine(comment+boolVal);
```

```
  comment = "cmBytes creates byte array [72,105,33]. String value of array: ";
  byte[] byteVal = iris.ClassMethodBytes(className,"cmBytes",72,105,33);
  Console.WriteLine(comment+ (new String(byteVal)));

  comment = "cmString() concatenates \"Hello\" + arg: ";
  string stringVal = iris.ClassMethodString(className,"cmString","World");
  Console.WriteLine(comment+stringVal);

  comment = "cmLong() returns the sum of two numbers: ";
  Long longVal = iris.ClassMethodLong(className,"cmLong",7,8);
  Console.WriteLine(comment+longVal);

  comment = "cmDouble() multiplies a number by 1.5: ";
  Double doubleVal = iris.ClassMethodDouble(className,"cmDouble",10);
  Console.WriteLine(comment+doubleVal);

  comment = "cmProcedure assigns a value to global array ^cmGlobal: ";
  iris.ClassMethodVoid(className,"cmVoid",67);
  // Read global array ^cmGlobal and then delete it
  Console.WriteLine(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);
  Console.WriteLine(comment+listVal.Get(1)+listVal.Get(2));
  } catch (Exception e){
    Console.WriteLine("method failed");
  }
```

ObjectScript Class User.NativeTest

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

Class Definition

```
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
```

3.2 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. ObjectScript also provides a special keyword to create method wrappers around existing routines without losing efficiency; set the CodeMode keyword equal to `call`.

The Native SDK methods in this section call user-defined ObjectScript functions or procedures and return a value of the type indicated by the method name: **FunctionBool()**, **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: int?, short?, string, long?, double?, float?, byte[], bool?, DateTime?, IRISList?, or IRISObject. If the connection is bidirection (see Using .NET Inverse Proxy Objects), then any .NET object can be used as an argument. See "Class IRIS Supported Datatypes" for more information about how the Native SDK handles these datatypes.

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 .NET 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 SDK

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): ";
Bool boolVal = iris.FunctionBool("fnBoolean",routineName,7,7);
Console.WriteLine(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));
Console.WriteLine(comment+(new String(byteVal)));

comment = "fnString() concatenates \"Hello\" + arg: ";
String stringVal = iris.FunctionString("fnString",routineName,"World");
Console.WriteLine(comment+stringVal);

comment = "fnLong() returns the sum of two numbers: ";
Long longVal = iris.FunctionInt("fnLong",routineName,7,8);
Console.WriteLine(comment+longVal);

comment = "fnDouble() multiplies a number by 1.5: ";
Double doubleVal = iris.FunctionDouble("fnDouble",routineName,5);
Console.WriteLine(comment+doubleVal);

comment = "fnProcedure assigns a value to global array ^fnGlobal: ";
```

```
iris.Procedure("fnProcedure",routineName,88);
// Read global array ^fnGlobal and then delete it
Console.WriteLine(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);
Console.WriteLine(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:

ObjectScript

```
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
```

3.3 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..

3.3.1 Using Pass-by-reference Arguments

The Native SDK supports pass by reference for both methods and functions. To pass an argument by reference, assign the argument value to an instance of class ADO.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 {
  Console.Write("\n\nCalling %SYS.DatabaseQuery.GetDatabaseFreeSpace()... ");
  status = iris.ClassMethodObject("%SYS.DatabaseQuery","GetDatabaseFreeSpace",dir,freeMB);
  Console.WriteLine("\nFree space in " + dir + " = " + freeMB.value + "MB");
}
catch (IRISException e) {
  Console.Write("Call to class method GetDatabaseFreeSpace() returned error:");
  Console.WriteLine(e.getMessage());
}
```

prints:

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

3.3.2 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 IRISException 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 an IRISException 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);
    Console.WriteLine("\nPassword validated!");

  } catch (IRISException e) {
    Console.WriteLine("Call to "+methodName+"(\""+pwd+"\") returned error:");
    Console.WriteLine(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 .NET Inverse Proxy Objects") the IRISObject.**InvokeStatusCode()** method can be used in exactly the same way.

4

Using .NET Inverse Proxy Objects

The .NET Native SDK is designed to take full advantage of .NET External Server connections, allowing completely transparent bidirectional communications between InterSystems IRIS and your .NET application.

Inverse proxy objects are .NET objects that allow you to control ObjectScript *target objects* over an external server gateway connection. You can use an inverse proxy object to call target methods and get or set target property values, manipulating the target object as easily as if it were a native .NET object.

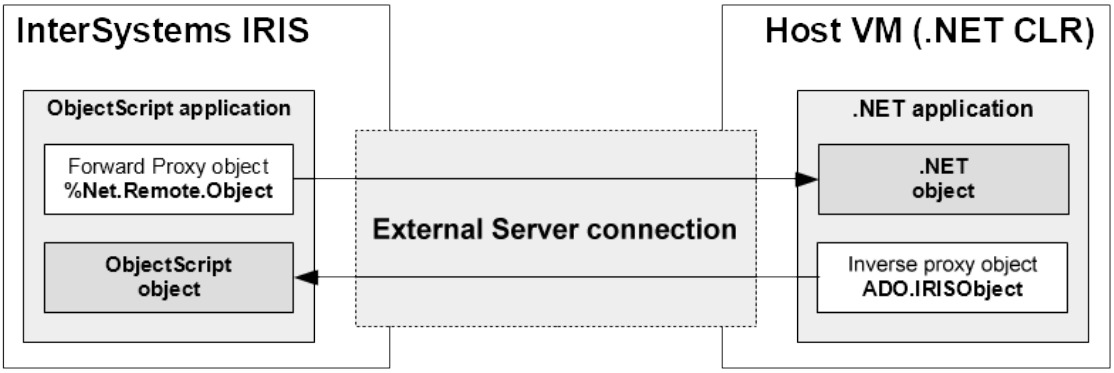
This section covers the following topics:

- Introducing External Servers — provides a brief overview of external servers.
- Creating an Inverse Proxy Object — describes methods used to create inverse proxy objects.
- Controlling a Target Object — demonstrates how inverse proxy objects are used.
- IRISObject Supported Datatypes — describes datatype-specific versions of the inverse proxy methods.

4.1 Introducing External Servers

External Server connections allow InterSystems IRIS target objects and .NET objects to interact freely, using the same connection and working together in the same context (database, session, and transaction). External server architecture is described in detail in *Using InterSystems External Servers*, but for the purposes of this discussion you can think of an external server connection as a simple black box allowing proxy objects on one side to control target objects on the other:

Figure 4–1: External Server connection



As the diagram shows, a *forward proxy object* is an ObjectScript proxy that controls a .NET object (see "Working with External Languages" in *Using InterSystems External Servers* for details). A .NET inverse proxy works in the other direction, allowing your .NET application to control an InterSystems IRIS target object.

4.2 Creating an Inverse Proxy Object

You can create an inverse proxy object by obtaining the OREF of an ObjectScript class instance (normally by calling the **%New()** method of the class) and casting it to IRISObject. (See "Calling ObjectScript Methods and Functions" for more information). The following methods can be used to generate an inverse proxy object:

- ADO.IRIS.**ClassMethodObject()** calls an ObjectScript class method and returns the result as an instance of object.
- ADO.IRIS.**FunctionObject()** calls an ObjectScript function and returns the result as an instance of object.

If the **%New()** method successfully creates a new target instance, an inverse proxy object will be generated for the instance. For example, the following call creates an inverse proxy object named *test* that controls an instance of ObjectScript class Demo.Test:

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

- **ClassMethodObject()** calls the **%New()** method of an ObjectScript class named Demo.Test to create a new target instance of that class.
- If the call to **%New()** returns a valid instance of the class, an inverse proxy for the new instance is generated, and **classMethodObject()** returns it as an Object.
- In .NET, the Object is cast to IRISObject, creating inverse proxy variable *test*.

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

4.3 Controlling a Target Object

An inverse proxy object is an instance of IRISObject. It provides methods **Invoke()** and **InvokeVoid()** to call target instance methods, and accessors **Get()** and **Set()** to read and write target properties. The example in this section uses an inverse proxy to control a target instance of ObjectScript class Demo.Test, which includes declarations for methods **initialize()** and **add()**, and property *name*:

Method and Property Declarations in 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 an inverse proxy object named *test* for a new instance of Demo.Test (as described in the previous section). The rest of the code uses *test* to control the target Demo.Test instance.

Controlling an instance of Demo.Test with an inverse 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", "Test One");

// 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.

4.4 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 supported datatypes.

IRISObject set() and get() methods

The IRISObject.**Set()** method accepts any .NET 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: **GetBool()**, **GetBytes()**, **GetDouble()**, **GetIRISList()**, **GetLong()**, **GetObject()** and **GetString()**.

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: **InvokeBool()**, **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 int?, short?, string, long?, double?, float?, byte[], bool?, DateTime?, IRISList?, or IRISObject. If the connection is bidirectional, any .NET 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.

5

Working with Global Arrays

The .NET Native SDK provides mechanisms for working with global arrays from your instance of InterSystems IRIS. This chapter covers the following topics:

- Introduction to Global Arrays — introduces global array concepts and provides a simple demonstration of how the .NET Native SDK 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: **Creating a .NET Connection**

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 .NET connection and creates an instance of IRIS:

```
//Open a connection to the server and create an IRIS object
        IRISConnection conn = new IRISConnection();
        conn.ConnectionString = "Server = localhost; "
        + "Port = 1972; " + "Namespace = USER; "
        + "Password = SYS; " + "User ID = _SYSTEM;";
        conn.Open();
        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 .NET connections, see Establishing .NET Connections in *Using .NET with InterSystems Software*. .

5.1 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 SDK **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" later in this chapter.

The Native SDK 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 SDK 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 SDK for .NET is part of the InterSystems.Data.IrisClient.dll library. For detailed information, see the Introduction to *Using .NET with InterSystems Software*.

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

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

//Open a connection to the server and create an IRIS object
        IRISConnection conn = new IRISConnection();
        conn.ConnectionString = "Server = localhost; "
        + "Port = 1972; " + "Namespace = USER; "
        + "Password = SYS; " + "User ID = _SYSTEM;";
        conn.Open();
        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");
        Console.WriteLine("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) {
        Console.WriteLine(e.Message);
```

```
        }
      }// end Main()
    } // end class NativeDemo
}
```

NativeDemo prints the following lines:

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

In this example, an IRISConnection object named *conn* provides a connection to the database associated with the USER namespace. Native SDK 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.

5.1.1 Glossary of Native SDK 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 SDK 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.

5.1.2 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”).
- 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.

5.2 Creating, Accessing, and Deleting Nodes

The Native SDK 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()**.

5.2.1 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.

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.

5.2.2 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: int?, short?, string, long?, double?, float?, byte[], bool?, DateTime?, IRISList?, and instances of objects that implement System.IO.MemoryStream (stored and retrieved as byte[]). A null value translates to ""

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

- Numeric datatypes: **GetBool()**, **GetInt16()**, **GetInt32()**, **GetInt64()**, **GetSingle()**, **GetDouble()**
- String datatypes: **GetString()**, **GetBytes()**, **GetIRISList()**
- Other datatypes: **GetDateTime()**, **GetObject()**

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

5.2.3 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.

5.3 Finding Nodes in a Global Array

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

- Iterating Over a Set of Child Nodes — describes how to iterate over all child nodes under a given parent node.
- Iteration in Conditional Loops — describes methods and properties that provide more control over iteration.

- Testing for Child Nodes and Node Values — describes how to find all subnodes regardless of node level, and identify which nodes have values.

5.3.1 Iterating Over a Set of Child Nodes

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

The heroes global array

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

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

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

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

Read child node values in reverse order

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

```
// Iterate in reverse, seeking nodes lower than heroes('dogs','V') in collation order
    IRISIterator iterDogs = iris.GetIRISReverseIterator("heroes","dogs");
    iterDogs.StartFrom("V");
    String output = "\nDog birth years: ";
    foreach (int BirthYear in iterDogs) {
        output += BirthYear + " ";
    };
    Console.WriteLine(output);
```

This code prints the following output:

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

The example does the following:

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

Two subnodes of *heroes("dogs")* are ignored:

- Child node *heroes("dogs","Whitefang")* will not be found because it is outside of the search range (`Whitefang` is higher than `V` in collation order).
- Level 3 node *heroes("dogs","Lassie","Timmy")* will not be found because it is a child of `Lassie`, not `dogs`.

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

Note: **Collation Order**

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

5.3.2 Iteration in Conditional Loops

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

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

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

Search for values that match items in a list

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

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

This code prints the following output:

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

The example does the following:

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

5.3.3 Testing for Child Nodes and Node Values

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

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

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

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

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

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

The following example consists of two methods:

- **TestNode()** will be called for each node in the *heroes* global array. It calls **IsDefined()** on the current node, and returns a boolean value indicating whether the node has child nodes. It also checks to see if the current subscript is `Timmy` or `Dolly`, and prints a message if so.

- **FindAllHeroes()** uses the return value of **TestNode()** to navigate the entire global array. It starts by iterating over the child nodes of root node *heroes*. Whenever **TestNode()** indicates that the current node has child nodes, **FindAllHeroes()** creates a new iterator to test the lower level child nodes.

Method FindAllHeroes()

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

```
public void FindAllHeroes() {
  string root = "heroes";

// Iterate over child nodes of root node heroes
  IRISIterator iterRoot = iris.GetIRISIterator(root);
  foreach (object node in iterRoot) {
    object sub1 = iterRoot.CurrentSubscript;
    bool hasChild1 = TestNode(iterRoot,sub1);

    // Process current child of heroes(sub1)
    if (hasChild1) {
      IRISIterator iterOne = iris.GetIRISIterator(root,sub1);
      foreach (object node in iterOne) {
        object sub2 = iterOne.CurrentSubscript;
        bool hasChild2 = TestNode(iterOne,sub1,sub2);

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

Method TestNode()

```
  public bool TestNode(IRISIterator iter, string root, params object[] subscripts) {
  // Test for values and child nodes
    int state = iris.IsDefined(root,subscripts);
    bool hasValue = (state%10 > 0); // has value if state is 1 or 11
    bool hasChild = (state > 9);    // has child if state is 10 or 11

  // Look for lost heroes


//    string[] lost = {"Timmy","Dolly"};
    var lost = new List<string> {"Timmy","Dolly"};

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

    return hasChild;
  }
```

5.4 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 bool, byte[], Single, Double, DateTime, Int16, Int32, Int64, String, IRISList, and instances of objects that implement System.IO.MemoryStream. 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 .NET variable: **GetBool()**, **GetSingle()**, **GetDouble()**, **GetInt16()**, **GetInt32()**, or **GetInt64()**. Given an integer node value, all numeric methods return meaningful values. Integer getters cannot reliably retrieve Single or Double values, and may return an inaccurate or meaningless value.

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 attempt to coerce it to the desired format.

The string getters assume that a 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 .NET classes

The IRIS class also supports getters for some .NET classes:

- **GetDateTime()** — gets instances of System.DateTime.
- **GetObject()** — returns the value of the target node as an object.
- **GetBytes()** — can be used to get objects that implement System.IO.MemoryStream. An instance of MemoryStream will be stored as byte[] when set as a node value. Use **GetBytes()** to retrieve the value, then initialize a MemoryStream with the returned byte [] value.

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.

6

Transactions and Locking

The *Native SDK for .NET* provides transaction and locking methods that use the InterSystems IRIS transaction model, as described in the following sections:

- 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 Native SDK and ADO.NET Transaction Models**

DO NOT mix the Native SDK transaction model with the ADO.NET/SQL transaction model.

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

6.1 Controlling Transactions

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

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

The Native SDK for .NET 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());

  Console.WriteLine("Node values before rollback and commit:");
  for (int ii=1;ii<4;ii++) {
    Console.Write(globalName + "(" + ii + ") = ");
    if (iris.isDefined(globalName,ii) > 1) Console.WriteLine(iris.GetString(globalName,ii));
    else Console.WriteLine("<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
  Console.WriteLine("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) Console.WriteLine(iris.GetString(globalName,ii));
    else Console.WriteLine("<valueless>");
  }
// prints: Node values after the transaction is committed:
//         myGlobal(1) = firstValue
//         myGlobal(2) = <valueless>
//         myGlobal(3) = <valueless>
```

6.2 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 SDK transaction model provides a set of locking methods that correspond to ObjectScript commands. These methods must not be used with the ADO.NET/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, int timeout, string globalName, Object[] subscripts)
Unlock(string lockMode, string globalName, Object[] 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, or SE for shared and escalating. Default is empty string (exclusive and non-escalating)
- *timeout* — number of seconds to wait when attempting to acquire the lock

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*

7

Native SDK for .NET Quick Reference

This is a quick reference for the InterSystems IRIS Native SDK for .NET, providing information on the following extension classes in InterSystems.Data.IRISClient.ADO:

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

All of these classes are part of the InterSystems ADO.NET Managed Provider (InterSystems.Data.IRISClient.ADO). They access the database through a standard .NET connection, and can be used without any special setup or installation procedures.

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

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

7.1 Class IRIS

Class IRIS is a member of InterSystems.Data.IRISClient.ADO (the InterSystems ADO.NET Managed Provider).

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

7.1.1 IRIS Method Details

ClassMethodBool()

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

```
bool ClassMethodBool(string className, string methodName, params 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()

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

```
byte[] ClassMethodBytes(string className, string methodName, params 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()

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

```
double ClassMethodDouble(string className, string methodName, params 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()

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

```
IRISList ClassMethodIRISList(string className, string methodName, params 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()

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

```
long ClassMethodLong(string className, string methodName, params 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()

ADO.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 an inverse proxy object (an instance of IRISObject) for the referenced object. See "Using .NET Inverse Proxy Objects" for details and examples.

```
object ClassMethodObject(string className, string methodName, params 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()

ADO.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 an IRISException error containing the ObjectScript $Status error status number and message.

```
void ClassMethodStatusCode(string className, string methodName, params 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()

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

```
string ClassMethodString(string className, string methodName, params 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()

ADO.IRIS.**ClassMethodVoid()** calls an ObjectScript class method with no return value, passing zero or more arguments.

```
void ClassMethodVoid(string className, string methodName, params 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()

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

```
void Close()
```

CreateIRIS() [static]

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

```
static IRIS CreateIRIS(IRISADOConnection conn)
```

parameters:

- `conn` — an instance of IRISConnection.

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

FunctionBool()

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

```
bool FunctionBool(string functionName, string routineName, params 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()

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

```
byte[] FunctionBytes(string functionName, string routineName, params 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()

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

```
double FunctionDouble(string functionName, string routineName, params 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.

FunctionInt()

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

```
long FunctionInt(string functionName, string routineName, params 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()

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

```
IRISList FunctionIRISList(string functionName, string routineName, params 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()

ADO.IRIS.**FunctionLong()** calls an ObjectScript function, passing zero or more arguments and returning an instance of Long.

```
long FunctionLong(string functionName, string routineName, params 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()

ADO.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 an inverse proxy object (an instance of IRISObject) for the referenced object. See "Using .NET Inverse Proxy Objects" for details and examples.

```
object FunctionObject(string functionName, string routineName, params 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()

ADO.IRIS.**FunctionString()** calls an ObjectScript function, passing zero or more arguments and returning an instance of string.

```
string FunctionString(string functionName, string routineName, params 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]

ADO.IRIS.**GetAPIVersion()** returns the Native SDK version string.

```
static String GetAPIVersion()
```

GetBool()

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

```
bool GetBool(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetBytes()

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

```
byte[] GetBytes(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetDateTime()

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

```
DateTime GetDateTime(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetDouble()

ADO.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.

```
Double GetDouble(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetInt16()

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

```
Int16 GetInt16(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetInt32()

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

```
Int32 GetInt32(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetInt64()

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

```
Int64 GetInt64(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetIRISIterator()

ADO.IRIS.**GetIRISIterator()** returns an IRISIterator object (see "Class IRISIterator") for the specified node with search direction set to `FORWARD`. See "Iterating Over a Set of Child Nodes" for more information and examples.

```
IRISIterator GetIRISIterator(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetIRISList()

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

```
IRISList GetIRISList(String globalName, params 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.

GetIRISReverseIterator()

ADO.IRIS.**GetIRISReverseIterator()** returns an IRISIterator object (see "Class IRISIterator") for the specified node with search direction set to BACKWARD. See "Iterating Over a Set of Child Nodes" for more information and examples.

```
IRISIterator GetIRISReverseIterator(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetObject()

ADO.IRIS.**GetObject()** gets the value of the global as an Object (or `null` if node does not exist). See "Using .NET Inverse Proxy Objects" for details and examples.

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

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetServerVersion()

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

```
String GetServerVersion()
```

GetSingle()

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

```
Single GetSingle(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetString()

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

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

```
string GetString(string globalName, params object[] subscripts)
```

parameters:

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

See "Class IRIS Supported Datatypes" for related information.

GetTLevel()

ADO.IRIS.**GetTLevel()** gets the level of the current nested Native SDK 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.

```
int GetTLevel()
```

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

Increment()

ADO.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.

```
long Increment(long? value, string globalName, params object[] subscripts)
```

parameters:

- `value` — long 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()

ADO.IRIS.**IsDefined()** returns a value indicating whether the specified node exists and if it contains a value. See "Testing for Child Nodes and Node Values" for more information and examples.

```
int IsDefined(string globalName, params 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()

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

```
void Kill(string globalName, params object[] subscripts)
```

parameters:

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

Lock()

ADO.IRIS.**Lock()** locks the global for a Native SDK 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.

```
bool Lock(string lockMode, int timeout, string globalName, params 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` — number of seconds to wait when attempting to acquire the Lock.
- `globalName` — global name.
- `subscripts` — zero or more subscripts specifying the target node.

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

Procedure()

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

```
void Procedure(string procedureName, string routineName, params 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()

ADO.IRIS.**ReleaseAllLocks()** is a Native SDK transaction method that releases all locks associated with the session.

```
void ReleaseAllLocks()
```

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

Set()

ADO.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.

```
void Set(bool? value, string globalName, params object[] subscripts)
void Set(byte[] value, string globalName, params object[] subscripts)
void Set(DateTime? value, string globalName, params object[] subscripts)
void Set(Double? value, string globalName, params object[] subscripts)
void Set(Int16? value, string globalName, params object[] subscripts)
void Set(Int32? value, string globalName, params object[] subscripts)
void Set(Int64? value, string globalName, params object[] subscripts)
void Set(IRISList value, String globalName, params object[] subscripts)
void Set(object value, string globalName, params object[] subscripts)
void Set(Single? value, string globalName, params object[] subscripts)
void Set(string value, string globalName, params object[] subscripts)
void Set(System.IO.MemoryStream value, string globalName, params 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 .NET is translated to the null string character `$CHAR(0)` in ObjectScript. A null in .NET is translated to the empty string in ObjectScript. This translation is consistent with the way .NET handles these values.
- System.IO.MemoryStream — an instance of an object that implements MemoryStream will be stored as a byte[] when set as the global value. A null in .NET is translated to the empty string in ObjectScript. Use **getBytes()** to retrieve the value, then initialize a MemoryStream with the returned byte [] value.

TCommit()

ADO.IRIS.**TCommit()** commits the current Native SDK transaction.

```
void TCommit()
```

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

TRollback()

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

```
void TRollback()
```

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

TRollbackOne()

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

```
void TRollbackOne()
```

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

TStart()

ADO.IRIS.**TStart()** starts/opens a Native SDK transaction.

```
void TStart()
```

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

Unlock()

ADO.IRIS.**Unlock()** unlocks the global in a Native SDK transaction. This method performs an incremental Unlock, not the implicit Unlock-before-Lock feature that is also offered in ObjectScript.

```
void Unlock(string lockMode, string globalName, params 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 Native SDK transaction model, and is not compatible with ADO.NET/SQL transaction methods. Never mix the two transaction models. See "Transactions and Locking" for more information and examples.

7.2 Class IRISIterator

Class IRISIterator is a member of InterSystems.Data.IRISClient.ADO (the InterSystems ADO.NET Managed Provider).

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

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

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

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

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

7.2.1 IRISIterator Method and Property Details

Property Current

ADO.IRISIterator.**Current** gets the value of the node at the current iterator position. In a `foreach` loop, this value is also assigned to the current loop variable. See "Iteration in Conditional Loops" for more information and examples.

```
object Current [get]
```

Property CurrentSubscript

ADO.IRISIterator.**CurrentSubscript** gets the lowest level subscript for the node at the current iterator position. For example, if the iterator points to node *^myGlobal(23,"somenode")*, the returned value will be `"somenode"`. See "Iteration in Conditional Loops" for more information and examples.

```
object CurrentSubscript [get]
```

MoveNext()

ADO.IRISIterator.**MoveNext()** implements System.Collections.IEnumerator. It returns `true` if the next value was retrieved, `false` if there are no more values. See "Iteration in Conditional Loops" for more information and examples.

```
bool MoveNext()
```

Reset()

ADO.IRISIterator.**Reset()** can be called after completing a `foreach` loop to reset the iterator to its starting position, allowing it to be used again. See "Iteration in Conditional Loops" for more information and examples.

```
void Reset()
```

StartFrom()

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

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

7.3 Class IRISList

Class IRISList is a member of InterSystems.Data.IRISClient.ADO (the InterSystems ADO.NET Managed Provider). It implements a .NET interface for InterSystems $LIST serialization. In addition to the IRISList constructors (described in the

following section), it is also returned by the following ADO.IRIS methods: **classMethodIRISList()**, **functionIRISList()**, **getIRISList()**.

7.3.1 IRISList Constructors

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

```
public IRISList()
public IRISList(IRISList list)
public 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.

7.3.2 IRISList Method Details

Add()

ADO.IRISList.**Add()** appends an Object of any supported type to the end of the IRISList. If *value* is an IRISList, it will be added as a single element (IRISList instances are never concatenated).

```
void Add(Object value)
```

parameters:

- `value` — Object value. The following types are supported: Int16, Int32, Int64, bool, Single, Double, string, byte[], IRISList.

Adding an IRISList element

Add() always appends an IRISList instance as a single object. However, you can use IRISList.**ToArray()** to convert an IRISList to an array, and then call **Add()** on each element separately.

AddRange()

ADO.IRISList.**AddRange()** appends each element of a collection to the end of the list as it is returned by the collection iterator. An exception will be thrown if any element is not one of the supported types.

```
void AddRange(System.Collections.IList list)
```

parameters:

- `list` — a collection containing only elements of the following types: Int16, Int32, Int64, bool, Single, Double, string, byte[], IRISList.

See "Class IRIS Supported Datatypes" for related information.

Clear()

ADO.IRISList.**Clear()** resets the list by removing all elements from the list.

```
void Clear()
```

Count()

ADO.IRISList.**Count()** iterates over the list and returns the number of elements encountered.

```
int Count()
```

DateToHorolog() [static]

ADO.IRISList.**DateToHorolog()** converts the *date* property of a DateTime value to an int representing the *day* field of a **$Horolog** string. Also see **HorologToDate()**.

```
static int DateToHorolog(DateTime value)
```

parameters:

- `value` — DateTime value to convert.

DateToPosix() [static]

ADO.IRISList.**DateToPosix()** converts a Time object to the *time* field of a **$Horolog** string.

```
static long DateToPosix(DateTime Value)
```

parameters:

- `Value` — DateTime value to convert.

Equals()

ADO.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.

```
override bool Equals(Object list)
```

parameters:

- `list` — instance of IRISList to compare.

Get()

ADO.IRISList.**Get()** returns the element at *index* as Object. Throws IndexOutOfRangeException 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()

ADO.IRISList.**GetList()** gets the element at *index* as an IRISList. Throws IndexOutOfRangeException 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

Throws IndexOutOfRangeException if the index is less then 1 or past the end of the list.

HorologToDate() [static]

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

```
static DateTime HorologToDate(int HorologValue)
```

parameters:

- `HorologValue` — int representing the *day* field of a **$Horolog** string.

HorologToTime() [static]

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

```
static TimeSpan HorologToTime(int HorologValue)
```

parameters:

- `HorologValue` — int representing the *time* field of a **$Horolog** string. The *date* field will be zeroed out (set to the epoch), so a TimeSpan with a milliseconds value outside the range `0L` to `86399999L` does not round trip.

PosixToDate() [static]

ADO.IRISList.**PosixToDate()** converts a %Library.PosixTime value to a DateTime. Also see **DateToPosix()**.

```
static DateTime PosixToDate(long PosixValue)
```

parameters:

- `PosixValue` — the %PosixTime value (a 64-bit signed integer).

Remove()

ADO.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.

```
bool Remove(int index)
```

parameters:

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

Set()

ADO.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)
```

Throws IndexOutOfRangeException if *index* is less than 1. All elements must be of the following types: Int16, Int32, Int64, bool, Single, Double, string, byte[], IRISList.

parameters:

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

Size()

ADO.IRISList.**Size()** returns the byte length of the serialized value for this IRISList.

```
int Size()
```

SubList()

ADO.IRISList.**SubList()** returns a new IRISList containing the elements in the closed range [*from*, *to*]. Throws IndexOutOfRangeException 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.

TimeToHorolog() [static]

ADO.IRISList.**TimeToHorolog()** converts a DateTime value to the *time* field of a **$Horolog** value. Also see **HorologToTime()**.

```
static int TimeToHorolog(DateTime value)
```

parameters:

- `value` — DateTime to be converted.

ToArray()

ADO.IRISList.**ToArray()** returns an array of the given type containing all of the elements in this list. If the list is empty, a zero length array will be returned.

IMPORTANT: This method should be used only if all elements can be coerced to be of the same type.

```
Object[] ToArray()
```

ToList()

ADO.IRISList.**ToList()** returns a List of the given type containing all of the elements in this IRISList. If the list is empty, a zero length List will be returned.

```
List<Object> ToList()
```

ToString()

ADO.IRISList.**ToString()** returns a printable representation of the list.

```
override 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

7.4 Class IRISObject

Class IRISObject is a member of InterSystems.Data.IRISClient.ADO (the InterSystems ADO.NET Managed Provider). It provides methods to work with Gateway inverse proxy objects (see "Using .NET Inverse 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:

- ADO.IRIS.**ClassMethodObject()**
- ADO.IRIS.**FunctionObject()**

If the called method or function returns an object that is a valid OREF, an inverse 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 .NET Inverse Proxy Objects" for details and examples.

7.4.1 IRISObject Method Details

Dispose()

ADO.IRISObject.**Dispose()** releases this instance of IRISObject.

```
void Dispose()
```

Get()

ADO.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.

GetBool()

ADO.IRISObject.**GetBool()** returns a property value of the proxy object as an instance of bool.

```
bool GetBool(string propertyName)
```

parameters:

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

See "IRISObject Supported Datatypes" for related information.

GetBytes()

ADO.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()

ADO.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()

ADO.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()

ADO.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()

ADO.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()

ADO.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()

ADO.IRISObject.**Invoke()** invokes an instance method of the object, returning value as object.

```
object Invoke(string methodName, params object[] args)
```

parameters:

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

Arguments may be of any of the following types: int?, short?, string, long?, double?, float?, byte[], bool?, DateTime?, IRISList?, or IRISObject. If the connection is bidirection, then any .NET object can be used as an argument.

See "IRISObject Supported Datatypes" for valid argument types and related information.

InvokeBoolean()

ADO.IRISObject.**InvokeBool()** invokes an instance method of the object, returning value as bool.

```
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 valid argument types and related information.

InvokeBytes()

ADO.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 valid argument types and related information.

InvokeDouble()

ADO.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 valid argument types and related information.

InvokeIRISList()

ADO.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 valid argument types and related information.

InvokeLong()

ADO.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 valid argument types and related information.

InvokeObject()

ADO.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 valid argument types and related information.

InvokeStatusCode()

ADO.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 valid argument types and related information.

InvokeString()

ADO.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 valid argument types and related information.

InvokeVoid()

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

```
void InvokeVoid(string methodName, params 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 valid argument types and 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()

ADO.IRISObject.**Set()** sets a property of the proxy object.

```
void Set(string propertyName, Object value)
```

parameters:

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