



Using Java with InterSystems Software

Version 2019.4
2020-01-28

Using Java with InterSystems Software

InterSystems IRIS Data Platform Version 2019.4 2020-01-28

Copyright © 2020 InterSystems Corporation

All rights reserved.

InterSystems, InterSystems IRIS, InterSystems Caché, InterSystems Ensemble, and InterSystems HealthShare are registered trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

About This Book	1
1 InterSystems Java Connectivity Options	3
1.1 Core Data Access APIs	3
1.2 InterSystems Object Gateway	5
1.3 InterSystems SQL Gateway	5
1.4 Third Party Framework Support	5
2 Using the JDBC Driver	7
2.1 Defining a JDBC Connection URL	7
2.1.1 Required Parameters	7
2.1.2 Optional Parameters	8
2.1.3 Setting the Port Parameter at the Command Line	8
2.1.4 Alternate Username and Password Parameters	8
2.2 Establishing JDBC Connections	9
2.2.1 Using IRISDataSource to Connect	9
2.2.2 Using DriverManager to Connect	9
2.2.3 Shared Memory Connections	10
2.2.4 Using a Connection Pool	10
2.3 Connection Properties	11
2.3.1 Listing Connection Properties	13
2.4 JDBC Logging	13
3 Configuration and Requirements	15
3.1 The InterSystems Java Class Packages	15
3.2 Client-Server Configuration	16
3.2.1 Java Client Requirements	16
3.2.2 InterSystems IRIS Server Configuration	16
4 JDBC for Occasional Users	17
4.1 A Simple JDBC Application	17
4.2 Using Queries	18
4.2.1 Executing a Prepared Statement	18
4.2.2 Using Callable Statements to Execute Stored Procedures	19
4.2.3 Returning Multiple Result Sets	19
4.3 Inserting and Updating Data	20
4.3.1 Inserting Data and Retrieving Generated Keys	20
4.3.2 Scrolling a Result Set	21
4.3.3 Using Transactions	22
5 JDBC Quick Reference	25
5.1 Class ConnectionPoolDataSource	25
5.2 Class IRISDataSource	26

About This Book

InterSystems IRIS® provides a wide variety of robust [Java connectivity options](#):

- Lightweight APIs that provide database access via Java objects or InterSystems multidimensional storage.
- Gateways that give InterSystems IRIS server applications direct access to Java resources and external databases.
- Implementations of third party APIs for Hibernate and Apache Spark.

All of these Java solutions are underpinned by the InterSystems JDBC driver, a powerful Type 4 (pure Java) fully compliant implementation of the JDBC API, closely coupled to InterSystems IRIS for maximum speed and efficiency.

The first chapter of this book is a survey of all Java solutions:

- [InterSystems Java Connectivity Options](#) provides an overview of all InterSystems IRIS Java technologies enabled by the JDBC driver.

The rest of the book provides detailed information on how to use the JDBC driver itself:

- [Using the JDBC Driver](#) gives a detailed description of the various ways to establish a JDBC connection to InterSystems IRIS or an external database.
- [Configuration and Requirements](#) provides details about client configuration and the InterSystems Java class packages.
- [JDBC for Occasional Users](#) is a quick overview of core JDBC API usage for readers not already familiar with Java database development.
- [JDBC Quick Reference](#) describes InterSystems-specific extension methods.

Related Documents

The following documents contain detailed information on Java solutions provided by InterSystems IRIS:

- [Persisting Java Objects with InterSystems XEP](#) describes how to use the Event Persistence API (XEP) for rapid Java object persistence.
- [Using the Native API for Java](#) describes how to use the Native API to access multidimensional storage (globals).
- [Using the InterSystems Spark Connector](#) describes how to use the InterSystems IRIS implementation of the Apache Spark Data Source API.
- “[InterSystems JDBC Support](#)” in [Implementation Reference for Java Third Party APIs](#) provides detailed information on InterSystems JDBC driver support and compliance, including the level of support for all optional features and a list of all InterSystems IRIS-specific additional features.
- [Using the Simple Data Transfer Utility](#) describes an InterSystems Java command-line utility used to expedite data transfer from a JDBC datasource or a CSV file to a JDBC compliant database.

1

InterSystems Java Connectivity Options

InterSystems IRIS® provides a wide variety of robust Java connectivity options:

- [Core Data Access APIs](#) provide lightweight data access via relational tables, objects, or globals.
- [InterSystems Object Gateway](#) provides an easy way access and manipulate both ObjectScript and Java objects using the same context and connection.
- [InterSystems SQL Gateway](#) provides customized connections to external databases and Java applications through an SQL interface.
- [Third Party Framework Support](#) includes interface implementations for *Spark* and *Hibernate*.

The InterSystems JDBC driver is at the core of all InterSystems IRIS Java solutions. It is a powerful Type 4 (pure Java) fully compliant implementation of the JDBC API, closely coupled to InterSystems IRIS for maximum speed and efficiency.

1.1 Core Data Access APIs

The InterSystems JDBC driver supports three lightweight Java APIs that provide direct access to InterSystems IRIS databases via relational tables, objects, and multidimensional storage.

Important: The Core Data Access APIs and the [Object Gateway](#) form an integrated suite of utilities that can all share the same underlying JDBC connection context. Your application can use any combination of desired features from any part of the suite.

JDBC driver API for relational table access

The InterSystems JDBC driver (described in this book) provides SQL based access to relational tables. It supports the following features:

- relational access
 - store and query tables via SQL
 - stored JDBC tables can be accessed as InterSystems IRIS objects
- JDBC API optimized for InterSystems IRIS
 - fully implemented type 4 (pure Java) JDBC API
 - extensions for unique InterSystems IRIS property settings

- high speed batch reads
- automatic connection pooling
- support for [third party Java frameworks](#)
 - optimized Spark JDBC data source API
 - Hibernate support

Detailed information is provided in later chapters of this book, and in the “[JDBC Driver Support](#)” chapter of the *Implementation Reference for Java Third Party APIs*.

XEP API for object access

The InterSystems XEP API is designed for extremely fast acquisition of data objects in real time, and can also be used as a convenient general purpose ORM interface. It supports the following features:

- optimized for speed
 - ultra-high speed real-time data acquisition. It can acquire data many times faster than standard JDBC.
 - batch reads
 - fine control over data serialization
- object-based access
 - lightweight alternative to Hibernate
 - store and query objects (create/read/update/delete)
 - schema import and customization
 - mapping for most standard datatypes
 - stored objects can also be accessed as JDBC tables
- full process control
 - control indexing and fetch level
 - control transactions and locking

See *Persisting Java Objects with InterSystems XEP* for details.

Native API for multidimensional storage access

The InterSystems Native API provides direct access to the tree-based sparse arrays (known as global arrays) of the InterSystems multidimensional storage model. It supports the following features:

- directly access and manipulate global arrays
 - create and delete nodes
 - iterate over nodes and create/read/update/delete values
 - control transactions and locking
- call server-side ObjectScript code:
 - call methods and access properties from any compiled class
 - call functions or procedures from any compiled .mac file

See [Using the Native API for Java](#) for details.

1.2 InterSystems Object Gateway

The Object Gateway provides an easy way access and manipulate both ObjectScript and Java objects in the same context, using the same connection. The Gateway is completely reentrant, allowing Java applications to manipulate ObjectScript objects and ObjectScript applications to manipulate Java objects, both using the same connection context (database, session and transaction) and physical connection.

ObjectScript applications can also use ODBC as an alternate connectivity option (providing access to .NET objects and ADO) without any major changes in your ObjectScript code.

See [Using the Object Gateway](#) for detailed information on both Java and .NET options.

1.3 InterSystems SQL Gateway

The InterSystems SQL Gateway connects InterSystems IRIS to external databases via JDBC. Various wizards can be used to create links to tables, views, or stored procedures in external sources. This allows you to read and store data in the external database just as you would on InterSystems IRIS, using objects and/or SQL queries. You even can generate class methods that perform the same actions as corresponding external stored procedures.

Gateway applications are written in ObjectScript and run on the server. They can also use ODBC as an alternate connectivity option (providing access to .NET objects and ADO) without any major changes in your ObjectScript code.

See [Using the SQL Gateway](#) for detailed information on both JDBC and ODBC options.

1.4 Third Party Framework Support

Java frameworks such as Spark and Hibernate use JDBC to interact with databases, and include interfaces that can be implemented to take advantage of features unique to a specific database. InterSystems IRIS provides implementations of the Spark Data Source interface and the Hibernate Dialect interface.

Spark Data Source API

The InterSystems Spark Connector is an implementation of the Data Source API for Apache Spark. It is a plug-compatible replacement for the standard Spark jdbc data source, allowing the Spark data processing engine to make optimal use of InterSystems IRIS and its distributed data capabilities

See [Using the InterSystems Spark Connector](#) for details.

Hibernate Dialect

The InterSystems Hibernate Dialect is a fully compliant implementation of the Hibernate dialect interface, providing a customized interface between Hibernate and InterSystems IRIS. Like most major dialect implementations, it is included as part of the Hibernate distribution.

See the “[Hibernate Support](#)” chapter in the [Implementation Reference for Java Third Party APIs](#) for details.

2

Using the JDBC Driver

This chapter gives a detailed description of the various ways to establish a JDBC connection between your application and InterSystems IRIS.

- [Defining a JDBC Connection URL](#) — describes how to specify the parameters that define a JDBC connection.
- [Establishing JDBC Connections](#) — describes how to establish and control connections using `DriverManager` or `DataSource`, and how to control connection pooling.
- [Connection Properties](#) — lists and describes connection properties for the InterSystems JDBC driver.
- [JDBC Logging](#) — describes how to enable logging when testing JDBC applications.

2.1 Defining a JDBC Connection URL

A `java.sql.Connection` URL supplies the connection with information about the host address, port number, and namespace to be accessed. The InterSystems JDBC driver also allows you to use several optional parameters.

2.1.1 Required Parameters

The minimal required URL syntax is:

```
jdbc:IRIS://<host>:<port>/<namespace>
```

where the parameters are defined as follows:

- *host* — IP address or Fully Qualified Domain Name (FQDN). For example, both `127.0.0.1` and `localhost` indicate the local machine.
- *port* — TCP port number on which the InterSystems IRIS SuperServer is listening. The default is `51773` (or the first available number higher than that if more than one instance of InterSystems IRIS is installed — see [DefaultPort](#) in the *Parameter File Reference*).
- *namespace* — InterSystems IRIS namespace to be accessed.

For example, the following URL specifies *host* as `127.0.0.1`, *port* as `51773`, and *namespace* as `User`:

```
jdbc:IRIS://127.0.0.1:51773/User
```

2.1.2 Optional Parameters

In addition to *host*, *port*, and *namespace*, you can also specify several optional parameters. The full syntax is:

```
jdbc:IRIS://<host>:<port>/<namespace>/<logfile>:<eventclass>:<nodelay>:<ssl>
```

where the optional parameters are defined as follows:

- *logfile* — specifies a JDBC log file (see “[JDBC Logging](#)”).
- *eventclass* — sets the transaction Event Class for this [IRISDataSource](#) object.
- *nodelay* — sets the TCP_NODELAY option if connecting via an [IRISDataSource](#) object. Toggling this flag can affect the performance of the application. Valid values are `true` and `false`. If not set, it defaults to `true`.
- *ssl* — enables SSL/TLS for both [IRISDriver](#) and [IRISDataSource](#) (see “[Using SSL/TLS with InterSystems IRIS](#)” in the *Security Administration Guide*). Valid values are `true` and `false`. If not set, it defaults to `false`.

Each of these optional parameters can be defined individually, without specifying the others. For example, the following URL sets only the required parameters and the *nodelay* option:

```
jdbc:IRIS://127.0.0.1:51773/User/::false
```

Other connection properties can be specified by passing them to `DriverManager` in a `Properties` object (see “[Using DriverManager to Connect](#)”).

2.1.3 Setting the Port Parameter at the Command Line

The `com.intersystems.port` property can be used to set the `port` parameter of the URL at the command line. Even if a program hard-codes the port number in the connection string, it can be changed in the command line. For example, assume that program `myJdbcProgram` sets the port to 51773 in a hard-coded connection string. The following command line will still allow it to run on port 9523:

```
java -cp ../lib/intersystems-jdbc-3.0.0.jar -Dcom.intersystems.port=9523 myJdbcProgram
```

The current value of this property can be retrieved programmatically with the following code:

```
String myport = java.lang.System.getProperty ("com.intersystems.port");
```

2.1.4 Alternate Username and Password Parameters

For the preferred ways to specify username and password, see “[Using IRISDataSource to Connect](#)” and “[Using DriverManager to Connect](#)” in the following section. However, it is also possible to specify the username and password in the URL string, although this is discouraged.

If password and username are supplied as part of the URL string, they will be used in order to connect. Otherwise, other mechanisms already in place will be invoked. The syntax is:

```
jdbc:IRIS://<host>:<port>/<namespace>/<options>?username=<string1>&password=<string2>
```

For example, the following URL string sets the required parameters, the *nodelay* option, and then the username and password:

```
"jdbc:IRIS://127.0.0.1:51773/User/::false?username=_SYSTEM&password=SYS"
```

The username and password strings are case sensitive.

2.2 Establishing JDBC Connections

This section describes how to establish and control connections using `DriverManager` or `DataSource`, and how to control connection pooling.

- [Using IRISDataSource to Connect](#) — describes using `IRISDataSource` to load the driver and create a `java.sql.Connection` object.
- [Using DriverManager to Connect](#) — describes using the `DriverManager` class to create a connection.
- [Shared Memory Connections](#) — describes a connection option used when the server and client are on the same machine.
- [Using a Connection Pool](#) — describes using the `IRISConnectionPoolDataSource` class to control the connection pool for your Java client applications.

2.2.1 Using IRISDataSource to Connect

Use `com.intersystems.jdbc.IRISDataSource` to load the driver and then create the `java.sql.Connection` object. This is the preferred method for connecting to a database and is fully supported by InterSystems IRIS.

Opening a connection with IRISDataSource

The following example loads the driver, and then uses `IRISDataSource` to create the connection and specify username and password:

```
try{
    IRISDataSource ds = new IRISDataSource();
    ds.setURL("jdbc:IRIS://127.0.0.1:51773/User");
    ds.setUser("_system");
    ds.setPassword("SYS");
    Connection dbconnection = ds.getConnection();
}
catch (SQLException e){
    System.out.println(e.getMessage());
}
catch (ClassNotFoundException e){
    System.out.println(e.getMessage());
}
```

This example deliberately uses the literal address, `127.0.0.1`, rather than `localhost`. On any system where the hostname resolves the same for IPv4 and IPv6, Java may attempt to connect via IPv6 if you use `localhost`.

Note: The `IRISDataSource` class provides an extended set of connection property accessors (such as `setURL()`, `setUser()` and `setPassword()` in this example). See “[Class IRISDataSource](#)” in the Quick Reference for a complete list of accessors, and see “[Connection Properties](#)” later in this chapter for more information on all connection properties.

2.2.2 Using DriverManager to Connect

The `DriverManager` class can also be used to create a connection. The following code demonstrates one possible way to do so:

```
Class.forName("com.intersystems.jdbc.IRISDriver").newInstance();
String url="jdbc:IRIS://127.0.0.1:51773/User";
String username = "_SYSTEM";
String password = "SYS";
dbconnection = DriverManager.getConnection(url,username,password);
```

You can also specify connection properties for `DriverManager` in a `Properties` object, as demonstrated in the following code:

```
String url="jdbc:IRIS://127.0.0.1:51773/User";
java.sql.Driver drv = java.sql.DriverManager.getDriver(url);

java.util.Properties props = new Properties();
props.put("user",username);
props.put("password",password);
java.sql.Connection dbconnection = drv.connect(url, props);
```

See “[Connection Properties](#)” later in this chapter for a complete list of connection properties provided by the InterSystems JDBC driver.

2.2.3 Shared Memory Connections

The standard JDBC connection to a remote InterSystems IRIS instance is over TCP/IP. To maximize performance, InterSystems IRIS also offers a shared memory connection for Java applications running on the same machine as an InterSystems IRIS instance. This connection avoids potentially expensive calls into the kernel network stack, providing optimal low latency and high throughput for JDBC operations.

If a connection specifies server address `localhost` or `127.0.0.1`, shared memory will be used by default. TCP/IP will be used if the actual machine address is specified. The connection will automatically fall back to TCP/IP if the shared memory device fails or is not available.

Shared memory can be disabled in the connection string by setting the *SharedMemory* property to `false`. The following example creates a connection that will not use shared memory even though the server address is specified as `127.0.0.1`:

```
Properties props = new Properties();
props.setProperty("SharedMemory", "false");
props.setProperty("user", "_system");
props.setProperty("password", "SYS");
IRISConnection conn = (IRISConnection)DriverManager.getConnection("jdbc:IRIS://127.0.0.1:51773/USER/",props);
```

Accessors `DataSource.getSharedMemory()` and `DataSource.setSharedMemory()` can be used to read and set the current connection mode. The `IRISConnection.isUsingSharedMemory()` method can also be used to test the connection mode.

Shared memory is not used for SSL or Kerberos connections. The JDBC log will include information on whether a shared memory connection was attempted and if it was successful (see “[JDBC Logging](#)”).

Note: **Shared memory connections do not work across container boundaries**

InterSystems does not currently support shared memory connections between two different containers. If a client tries to connect across container boundaries using `localhost` or `127.0.0.1`, the connection mode will default to shared memory, causing it to fail. This applies regardless of whether the Docker `--network host` option is specified. You can guarantee a TCP/IP connection between containers either by specifying the actual hostname for the server address, or by disabling shared memory in the connection string (as demonstrated above).

Shared memory connections can be used without problems when the server and client are in the same container.

2.2.4 Using a Connection Pool

The `com.intersystems.jdbc.IRISConnectionPoolDataSource` class implements the `javax.sql.ConnectionPoolDataSource` interface, providing a connection pool for your Java client applications.

Note: This implementation is intended only for testing and development. It should not be used in production.

Here are the steps for using a connection pool with InterSystems IRIS:

1. Import the needed packages:

```
import com.intersystems.jdbc.*;
import java.sql.*;
```

2. Instantiate an `IRISConnectionPoolDataSource` object. Use the `restart()` method to close all of the physical connections and empty the pool. Use `setURL()` to set the database URL (see [Defining a JDBC Connection URL](#)) for the pool's connections.

```
IRISConnectionPoolDataSource pds = new IRISConnectionPoolDataSource();
pds.restartConnectionPool();
pds.setURL("jdbc:IRIS://127.0.0.1:51773/User");
pds.setUser("_system");
pds.setPassword("SYS");
```

3. Initially, `getPoolCount` returns 0.

```
System.out.println(pds.getPoolCount()); //outputs 0.
```

4. Use `IRISConnectionPoolDataSource.getConnection()` to retrieve a database connection from the pool.

```
Connection dbConnection = pds.getConnection();
```

CAUTION: InterSystems IRIS driver connections must always be obtained by calling the `getConnection()` method (inherited from `IRISDataSource`). Do not use the `getPooledConnection()` methods, which are for use only within the InterSystems IRIS driver.

5. Close the connection. Now `getPoolCount` returns 1.

```
dbConnection.close();
System.out.println(pds.getPoolCount()); //outputs 1
```

2.2.4.1 Statement Pooling

JDBC 4.0 adds an additional infrastructure, statement pooling, which stores optimized statements in a cache the first time they are used. Statement pools are maintained by connection pools, allowing pooled statements to be shared between connections. All the implementation details are completely transparent to the user, and it is up to the driver to provide the required functionality.

InterSystems JDBC implemented statement pooling long before the concept became part of the JDBC specification. While the InterSystems IRIS driver uses techniques similar to those recommended by the specification, the actual pooling implementation is highly optimized. Unlike most implementations, InterSystems JDBC has three different statement pooling caches. One roughly corresponds to statement pooling as defined by the JDBC specification, while the other two are InterSystems IRIS specific optimizations. See “[Cached Queries](#)” in the *SQL Optimization Guide* for an explanation of InterSystems IRIS statement caching. As required, InterSystems JDBC statement pooling is completely transparent to the user.

The InterSystems JDBC implementation supports Statement methods `setPoolable()` and `isPoolable()` as hints to whether the statement in question should be pooled. InterSystems IRIS uses its own heuristics to determine appropriate sizes for all three of its statement pools, and therefore does not support limiting the size of a statement pool by setting the `maxStatements` property in `IRISConnectionPoolDataSource`. The optional `javax.sql.StatementEventListener` interface is unsupported (and irrelevant) for the same reason.

2.3 Connection Properties

This section lists and describes the connection properties provided by `jdbc.IRISDataSource` (the InterSystems implementation of `javax.sql.DataSource`). Connection properties can be set by passing them to `DriverManager` (as described in “[Using DriverManager to Connect](#)”) or calling connection property accessors (see “[Class IRISDataSource](#)” in the reference chapter for a complete list).

The following connection properties are supported:

connection security level

Optional. Integer indicating Connection Security Level. Valid levels are 0, 1, 2, 3, or 10. Default = 0.

0 - Instance Authentication (Password)

1 - Kerberos (authentication only)

2 - Kerberos with Packet Integrity

3 - Kerberos with Encryption

10 - SSL/TLS

See IRISDataSource methods [getConnectionSecurityLevel\(\)](#) and [setConnectionSecurityLevel\(\)](#).

FeatureOption

Optional. Integer specifying fast select and fast insert settings for query optimization. Valid values are 0, 1, 2, and 3. Default=3.

0 - disable *Fast Select* and *Fast Insert*

1 - enable *Fast Select* but disable *Fast Insert*

2 - disable *Fast Select* but enable *Fast Insert*

3 - (default) enable both *Fast Select* and *Fast Insert*

See “Fast Select” in *Using InterSystems SQL* and “Fast Insert” in the *InterSystems SQL Reference*.

key recovery password

Optional. String containing current Key Recovery Password setting. Default = null. See IRISDataSource methods [getKeyRecoveryPassword\(\)](#) and [setKeyRecoveryPassword\(\)](#).

password

Required. String containing password. Default = null. See IRISDataSource methods [getPassword\(\)](#) and [setPassword\(\)](#)

service principal name

Optional. String indicating Service Principal Name. Default = null. See IRISDataSource methods [getServicePrincipalName\(\)](#) and [setServicePrincipalName\(\)](#)

SharedMemory

Optional. Boolean indicating whether or not to always use shared memory for localhost and 127.0.0.1. Default = null. See IRISDataSource methods [getSharedMemory\(\)](#) and [setSharedMemory\(\)](#). Also see “[Shared Memory Connections](#)” earlier in this chapter.

SO_RCVBUF

Optional. Integer indicating TCP/IP SO_RCVBUF value (ReceiveBufferSize). Default = 0 (use system default value).

SO_SNDBUF

Optional. Integer indicating TCP/IP SO_SNDBUF value (SendBufferSize). Default = 0 (use system default value).

SSL configuration name

Optional. String containing current SSL Configuration Name for this object. Default = `null`. See `IRISDataSource` methods `getSSLConfigurationName()` and `setSSLConfigurationName()`.

TCP_NODELAY

Optional. Boolean indicating TCP/IP `TCP_NODELAY` flag (Nodelay). Default = `true`.

Connection parameter `nodelay` — sets the `TCP_NODELAY` option if connecting via a `IRISDataSource` object. Toggling this flag can affect the performance of the application. Valid values are `true` and `false`. If not set, it defaults to `true`.

See `IRISDataSource` methods `getNodeDelay()` and `setNodeDelay()`

TransactionIsolationLevel

Optional. A `java.sql.Connection` constant indicating Transaction Isolation Level. Valid values are `TRANSACTION_READ_UNCOMMITTED` or `TRANSACTION_READ_COMMITTED`. Default = `null` (use system default value `TRANSACTION_READ_UNCOMMITTED`).

See `IRISDataSource` methods `getDefaultTransactionIsolation()` and `setDefaultTransactionIsolation()`

user

Required. String containing username. Default = `null`. See `IRISDataSource` methods `getUser()` and `setUser()`

2.3.1 Listing Connection Properties

Code similar to the following can be used to list the available properties for any compliant JDBC driver:

```
java.sql.Driver drv = java.sql.DriverManager.getDriver(url);
java.sql.Connection dbconnection = drv.connect(url, user, password);
java.sql.DatabaseMetaData meta = dbconnection.getMetaData();
System.out.println ("\n\nDriver Info: =====");
System.out.println (meta.getDriverName());
System.out.println ("release " + meta.getDriverVersion() + "\n");

java.util.Properties props = new Properties();
DriverPropertyInfo[] info = drv.getPropertyInfo(url,props);
for(int i = 0; i < info.length; i++) {
    System.out.println ("\n" + info[i].name);
    if (info[i].required) {System.out.print("  Required");}
    else {System.out.print("  Optional");}
    System.out.println (" , default = " + info[i].value);
    if (info[i].description != null)
        System.out.println ("  Description:" + info[i].description);
    if (info[i].choices != null) {
        System.out.println ("  Valid values: ");
        for(int j = 0; j < info[i].choices.length; j++)
            System.out.println("      " + info[i].choices[j]);
    }
}
```

2.4 JDBC Logging

If your applications encounter any problems, you can monitor by enabling logging. Run your application, ensuring that you trigger the error condition, then check all the logs for error messages or any other unusual activity. The cause of the error is often obvious.

Note: Enable logging only when you need to perform troubleshooting. You should not enable logging during normal operation, because it will dramatically slow down performance.

To enable logging for JDBC when connecting to InterSystems IRIS, add a log file name to the end of your JDBC connection string. When you connect, the driver will save a log file that will be saved to the working directory of the application.

For example, suppose your original connection string is as follows:

```
jdbc:IRIS://127.0.0.1:51773/USER
```

To enable logging, change this to the following and then reconnect:

```
jdbc:IRIS://127.0.0.1:51773/USER/myjdbc.log
```

This log records the interaction from the perspective of the InterSystems IRIS database.

If the specified log file already exists, new log entries will be appended to it by default. To delete the existing file and create a new one, prefix the log file name with a plus character (+). For example, the following string would delete myjdbc.log (if it exists) and create a new log file with the same name:

```
jdbc:IRIS://127.0.0.1:51773/USER/+myjdbc.log
```

3

Configuration and Requirements

To use the InterSystems JDBC driver, you should be familiar with the Java programming language and have some understanding of how Java is configured on your operating system. If you are performing custom configuration of the InterSystems JDBC driver on UNIX®, you should also be familiar with compiling and linking code, writing shell scripts, and other such tasks.

3.1 The InterSystems Java Class Packages

The main InterSystems Java class packages are contained in the following files:

- `intersystems-jdbc-3.0.0.jar` — the core JDBC jar file. All of the other files in this list are dependent on this file.
In addition to the core JDBC API, this file also includes the classes that implement the Native API (see [Using the Native API for Java](#)).
- `intersystems-xep-3.0.0.jar` — required for XEP Java persistence applications (see [Persisting Java Objects with InterSystems XEP](#)). Depends on the JDBC jar.
- `intersystems-gateway-3.0.0.jar` — required for Java Object Gateway applications (see [Using the Object Gateway](#)). Depends on the JDBC jar.
- `intersystems-uima-1.0.0.jar` — required for UIMA support (see [Using InterSystems UIMA](#)). Depends on the JDBC jar.
- `intersystems-spark-1.0.0.jar` — required for the InterSystems implementation of the Apache Spark Data Source API (see [Using the InterSystems Spark Connector](#)). Depends on the JDBC jar.
- `intersystems-utils-3.0.0.jar` — required for the JDBC Simple Data Transfer utility (see article “Using the Simple Data Transfer Utility”). Depends on the JDBC jar, and requires the XEP jar if XEP transfer mode is used.

There are separate versions of these files for each supported version of Java, located in subdirectories of `<install-dir>/dev/java/lib` (for example, `<install-dir>/dev/java/lib/JDK18` contains the files for Java 1.8).

You can determine the location of `<install-dir>` (the InterSystems IRIS root directory) for an instance of InterSystems IRIS by opening the InterSystems terminal in that instance and issuing the following ObjectScript command:

```
write $system.Util.InstallDirectory()
```

See “[InterSystems IRIS Installation Directory](#)” in the [Installation Guide](#) for system-specific information on the location of `<install-dir>`.

3.2 Client-Server Configuration

The Java client and InterSystems IRIS server may reside on the same physical machine or they may be located on different machines. Only the InterSystems IRIS server machine requires a copy of InterSystems IRIS; client applications do not require a local copy.

3.2.1 Java Client Requirements

The InterSystems IRIS Java client requires a supported version of the Java JDK. Client applications do not require a local copy of InterSystems IRIS.

The online [InterSystems IRIS Data Platform Supported Platforms](#) document for this release specifies the current requirements for all Java-based client applications. See the section on “[Supported Java Technologies](#)” for supported Java releases.

The core component of the Java binding is a file named `intersystems-jdbc-3.0.0.jar`, which contains the Java classes that provide the connection and caching mechanisms for communication with the InterSystems IRIS server and JDBC connectivity. Client applications do not require a local copy of InterSystems IRIS, but the `intersystems-jdbc-3.0.0.jar` file must be on the class path of the application when compiling or using Java proxy classes. See “[The InterSystems IRIS Java Class Packages](#)” for more information on these files.

3.2.2 InterSystems IRIS Server Configuration

Every Java client that wishes to connect to an InterSystems IRIS server needs a URL that provides the server IP address, TCP port number, and InterSystems IRIS namespace, plus a username and password.

To run a Java or JDBC client application, make sure that your installation meets the following requirements:

- The client must be able to access a machine that is currently running a compatible version of the InterSystems IRIS server (see the online [InterSystems IRIS Data Platform Supported Platforms](#) document for this release). The client and the server can be running on the same machine.
- Your class path must include the version of `intersystems-jdbc-3.0.0.jar` that correspond to the client version of the Java JDK (see “[The InterSystems IRIS Java Class Packages](#)”).
- To connect to the InterSystems IRIS server, the client application must have the following information:
 - The IP address of the machine on which the InterSystems IRIS Superserver is running. The Java sample programs use the address of the server on the local machine (`localhost` or `127.0.0.1`). If you want a sample program to connect to a different system you will need to change its connection string and recompile it.
 - The TCP port number on which the InterSystems IRIS Superserver is listening. The Java sample programs use 51773 (the default). If you want a sample program to use a different port you will need to change its connection string and recompile it.
 - A valid SQL username and password. You can manage SQL usernames and passwords on the System Administration > Security > Users page of the Management Portal. The Java sample programs use the administrator username, “`_SYSTEM`” and the default password “`SYS`” or “`sys`”. Typically, you will change the default password after installing the server. If you want a sample program to use a different username and password you will need to change it and recompile it.
 - The server namespace containing the classes and data that your client application will use.

See “[Establishing JDBC Connections](#)” for detailed information on connecting to the InterSystems IRIS server.

4

JDBC for Occasional Users

JDBC needs no introduction for experienced Java database developers, but it can be very useful even if you only use Java for occasional small utility applications. This chapter is a quick overview of JDBC that provides examples of Java code for querying databases and working with the results.

- [A Simple JDBC Application](#) is a complete but very simple application that demonstrates the basic features of JDBC.
- [Using Queries](#) is an overview of JDBC SQL query classes.
- [Inserting and Updating Data](#) demonstrates using JDBC result sets to insert and update data in an InterSystems IRIS database.

4.1 A Simple JDBC Application

This section describes a very simple JDBC application that demonstrates the use of some of the most common JDBC classes:

- An `IRISDataSource` object is used to create a `Connection` object that links the JDBC application to the InterSystems IRIS database.
- The `Connection` object is used to create a `PreparedStatement` object that can execute a dynamic SQL query.
- The `PreparedStatement` query returns a `ResultSet` object that contains the requested rows.
- The `ResultSet` object has methods that can be used to move to a specific row and read or update specified columns in the row.

All of these classes are discussed in more detail later in the chapter.

The SimpleJDBC Application

To begin, import the JDBC packages and open a `try` block:

```
import java.sql.*;
import javax.sql.*;
import com.intersystems.jdbc.*;

public class SimpleJDBC{
    public static void main() {
        try {

// Use IRISDataSource to open a connection
            Class.forName ("com.intersystems.jdbc.IRISDriver").newInstance();
            IRISDataSource ds = new IRISDataSource();
            ds.setURL("jdbc:IRIS://127.0.0.1:51773/User");
            Connection dbconn = ds.getConnection("_SYSTEM","SYS");
```

```
// Execute a query and get a scrollable, updatable result set.
String sql="Select Name from Demo.Person Order By Name";
int scroll=ResultSet.TYPE_SCROLL_SENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;
PreparedStatement pstmt = dbconn.prepareStatement(sql,scroll,update);
java.sql.ResultSet rs = pstmt.executeQuery();

// Move to the first row of the result set and change the name.
rs.first();
System.out.println("\n Old name = " + rs.getString("Name"));
rs.updateString("Name", "Bill. Buffalo");
rs.updateRow();
System.out.println("\n New name = " + rs.getString("Name") + "\n");

// Close objects and catch any exceptions.
pstmt.close();
rs.close();
dbconn.close();
} catch (Exception ex) {
System.out.println("SimpleJDBC caught exception: "
+ ex.getClass().getName() + ": " + ex.getMessage());
}
} // end main()
} // end class SimpleJDBC
```

Note: In the rest of this chapter, examples will be presented as fragments of code, rather than whole applications. These examples demonstrate some basic features as briefly and clearly as possible, and are not intended to teach good coding practices. It will be assumed that a connection has already been opened, and that all code fragments are within an appropriate `try/catch` statement.

4.2 Using Queries

The `sql.java` package provides the `PreparedStatement` and `CallableStatement` classes to query databases and return a `ResultSet`. Both classes are instantiated by calls to `Connection` methods. The following sections discuss how to use these classes:

- [Executing a Prepared Statement](#) — an example using implicit join syntax.
- [Executing Stored Procedures with CallableStatement](#) — an example that executes a stored procedure.
- [Returning Multiple Result Sets](#) — accessing multiple result sets returned by InterSystems IRIS stored procedures.

4.2.1 Executing a Prepared Statement

The following query uses a prepared statement to return a list of all employees with names beginning in “A” through “E” who work for a company with a name starting in “M” through “Z”:

```
Select ID, Name, Company->Name from Demo.Employee
Where Name < ? and Company->Name > ?
Order By Company->Name
```

Note: This statement uses [Implicit Join](#) syntax (the `->` operator), which provides a simple way to access the `Company` class referenced by `Demo.Employee`.

To implement the prepared statement:

- Create the string containing the query and use it to initialize the `PreparedStatement` object, then set the values of the query parameters and execute the query:

```
String sql=
  "Select ID, Name, Company->Name from Demo.Employee " +
  "Where Name < ? and Company->Name > ? " +
  "Order By Company->Name";
PreparedStatement pstmt = dbconnection.prepareStatement(sql);

pstmt.setString(1,"F");
pstmt.setString(2,"L");
java.sql.ResultSet rs = pstmt.executeQuery();
```

- Retrieve and display the result set:

```
java.sql.ResultSet rs = pstmt.executeQuery();
ResultSetMetaData rsmd = rs.getMetaData();
int colnum = rsmd.getColumnCount();
while (rs.next()) {
  for (int i=1; i<=colnum; i++) {
    System.out.print(rs.getString(i) + " ");
  }
  System.out.println();
}
```

4.2.2 Using Callable Statements to Execute Stored Procedures

The following code executes **ByName**, an InterSystems IRIS stored procedure contained in Demo.Person:

- Create a `java.sql.CallableStatement` object and initialize it with the name of the stored procedure. The `SqlName` of the procedure is **SP_Demo_By_Name**, which is how it must be referred to in the Java client code:

```
String sql="call Demo.SP_Demo_By_Name(?)"
CallableStatement cs = dbconnection.prepareCall(sql);
```

- Set the value of the query parameter and execute the query, then iterate through the result set and display the data:

```
cs.setString(1,"A");
java.sql.ResultSet rs = cs.executeQuery();

ResultSetMetaData rsmd = rs.getMetaData();
int colnum = rsmd.getColumnCount();
while (rs.next()) {
  for (int i=1; i<=colnum; i++)
    System.out.print(rs.getString(i) + " ");
}
System.out.println();
```

4.2.3 Returning Multiple Result Sets

InterSystems IRIS allows you to define a stored procedure that returns multiple result sets. The InterSystems JDBC driver supports the execution of such stored procedures. Here is an example of an InterSystems IRIS stored procedure that returns two result sets (note that the two query results have different column structures):

```
/// This class method produces two result sets.
ClassMethod DRS(st) [ ReturnResultsets, SqlProc ]
{
  $$$ResultSet("select Name from Demo.Person where Name %STARTSWITH :st")
  $$$ResultSet("select Name, DOB from Demo.Person where Name %STARTSWITH :st")
  Quit
}
```

\$\$\$ResultSet is a predefined InterSystems macro that prepares a SQL statement (specified as a string literal), executes it, and returns the resultset.

The following code executes the stored procedure and iterates through both of the returned result sets:

- Create the `java.sql.CallableStatement` object and initialize it using the name of the stored procedure. Set the query parameters and use **execute** to execute the query:

```
CallableStatement cs = dbconnection.prepareCall("call Demo.Person_DRS(?)");
cs.setString(1,"A");
boolean success=cs.execute();
```

- Iterate through the pair of result sets displaying the data. After **getResultSet** retrieves the current result set, **getMoreResults** closes it and moves to the CallableStatement object's next result set.

```
if(success) do{
    java.sql.ResultSet rs = cs.getResultSet();
    ResultSetMetaData rsmd = rs.getMetaData();
    for (int j=1; j<rsmd.getColumnCount() + 1; j++)
        System.out.print(rsmd.getColumnName(j)+ "\t\t");
    System.out.println();
    int colnum = rsmd.getColumnCount();
    while (rs.next()) {
        for (int i=1; i<=colnum; i++)
            System.out.print(rs.getString(i) + " \t ");
        System.out.println();
    }
    System.out.println();
} while (cs.getMoreResults());
```

4.3 Inserting and Updating Data

There are several ways to insert and update InterSystems IRIS data using JDBC:

- [Inserting Data and Retrieving Generated Keys](#) — using PreparedStatement and the SQL INSERT command.
- [Scrolling a Result Set](#) — randomly accessing any row of a result set.
- [Using Transactions](#) — using the JDBC transaction API to commit or roll back changes.

4.3.1 Inserting Data and Retrieving Generated Keys

The following code inserts a new row into Demo.Person and retrieves the generated ID key.

- Create the PreparedStatement object, initialize it with the SQL string, and specify that generated keys are to be returned:

```
String sqlIn="INSERT INTO Demo.Person (Name,SSN,DOB) " + "VALUES(?,?,?)";
int keys=Statement.RETURN_GENERATED_KEYS;
PreparedStatement pstmt = dbconnection.prepareStatement(sqlIn, keys);
```

- Set the values for the query parameters and execute the update:

```
String SSN = Demo.util.generateSSN(); // generate a random SSN
java.sql.Date DOB = java.sql.Date.valueOf("1973-02-01");

pstmt.setString(1,"Smith,John"); // Name
pstmt.setString(2,SSN); // Social Security Number
pstmt.setDate(3,DOB); // Date of Birth
pstmt.executeUpdate();
```

- Each time you insert a new row, the system automatically generates an object ID for the row. The generated ID key is retrieved into a result set and displayed along with the *SSN*:

```
java.sql.ResultSet rsKeys = pstmt.getGeneratedKeys();
rsKeys.next();
String newID=rsKeys.getString(1);
System.out.println("new ID for SSN " + SSN + " is " + newID);
```

Although this code assumes that the ID will be the first and only generated key in *rsKeys*, this is not always a safe assumption in real life.

- Retrieve the new row by ID and display it (*Age* is a calculated value based on *DOB*).


```
String sqlOut="SELECT IName,Age,SSN FROM Demo.Person WHERE ID="+newID;
pstmt = dbconnection.prepareStatement(sqlOut);
java.sql.ResultSet rsPerson = pstmt.executeQuery();

int colnum = rsPerson.getMetaData().getColumnCount();
rsPerson.next();
for (int i=1; i<=colnum; i++)
    System.out.print(rsPerson.getString(i) + " ");
System.out.println();
```

4.3.2 Scrolling a Result Set

The InterSystems JDBC driver supports scrollable result sets, which allow your Java applications to move both forward and backward through the resultset data. The **prepareStatement()** method uses following parameters to determine how the result set will function:

- The *resultSetType* parameter determines how changes are displayed:
 - `ResultSet.TYPE_SCROLL_SENSITIVE` creates a scrollable result set that displays changes made to the underlying data by other processes.
 - `ResultSet.TYPE_SCROLL_INSENSITIVE` creates a scrollable result set that only displays changes made by the current process.
- The *resultSetConcurrency* parameter must be set to `ResultSet.CONCUR_UPDATABLE` if you intend to update the result set.

The following code creates and uses a scrollable result set:

- Create a `PreparedStatement` object, set the query parameters, and execute the query:

```
String sql="Select Name, SSN from Demo.Person "+
    " Where Name > ? Order By Name";
int scroll=ResultSet.TYPE_SCROLL_SENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;

PreparedStatement pstmt = dbconnection.prepareStatement(sql,scroll,update);
pstmt.setString(1,"S");
java.sql.ResultSet rs = pstmt.executeQuery();
```

A result set that is going to have new rows inserted should not include the InterSystems IRIS ID column. ID values are defined automatically by InterSystems IRIS.

- The application can scroll backwards as well as forwards through this result set. Use **afterLast** to move the result set's cursor to after the last row. Use **previous** to scroll backwards.

```
rs.afterLast();
int colnum = rs.getMetaData().getColumnCount();
while (rs.previous()) {
    for (int i=1; i<=colnum; i++)
        System.out.print(rs.getString(i) + " ");
    System.out.println();
}
```

- Move to a specific row using **absolute**. This code displays the contents of the third row:

```
rs.absolute(3);
for (int i=1; i<=colnum; i++)
    System.out.print(rs.getString(i) + " ");
System.out.println();
```

- Move to a specific row relative to the current row using **relative**. The following code moves to the first row, then scrolls down two rows to display the third row again:

```
rs.first();
rs.relative(2);
for (int i=1; i<=colnum; i++)
    System.out.print(rs.getString(i) + " ");
System.out.println();
```

- To update a row, move the cursor to that row and update the desired columns, then invoke **updateRow()**:

```
rs.last();
rs.updateString("Name", "Avery. Tara R");
rs.updateRow();
```

- To insert a row, move the cursor to the “insert row” and then update that row's columns. Be sure that all non-nullable columns are updated. Finally, invoke **insertRow()**:

```
rs.moveToInsertRow();
rs.updateString(1, "Abelson,Alan");
rs.updateString(2, Demo.util.generateSSN());
rs.insertRow();
```

4.3.3 Using Transactions

The InterSystems JDBC driver supports the standard JDBC transaction model.

- In order to group SQL statements into a transaction, you must first disable autocommit mode using **setAutoCommit()**:

```
dbconnection.setAutoCommit(false);
```

- Use **commit()** to commit to the database all SQL statements executed since the last execution of **commit()** or **rollback()**:

```
pstmt1.execute();
pstmt2.execute();
pstmt3.execute();
dbconnection.commit();
```

- Use **rollback()** to roll back all of the transactions in a transactions. Here the **rollback()** is invoked if **SQLException** is thrown by any SQL statement in the transaction:

```
catch(SQLException ex) {
    if (dbconnection != null) {
        try {
            dbconnection.rollback();
        } catch (SQLException excep){
            // (handle exception)
        }
    }
}
```

Here is a brief summary of the `java.sql.Connection` methods used in this example:

- **setAutoCommit()**

By default `Connection` objects are in autocommit mode. In this mode an SQL statement is committed as soon as it is executed. To group multiple SQL statements into a transaction, first use `setAutoCommit(false)` to take the `Connection` object out of autocommit mode. Use `setAutoCommit(true)` to reset the `Connection` object to autocommit mode.

- **commit()**

Executing **commit()** commits all SQL statements executed since the last execution of either **commit()** or **rollback()**. Note that no exception will be thrown if you call **commit()** without first setting autocommit to `false`.

- **rollback()**

Executing **rollback** aborts a transaction and restores any values changed by the transaction back to their original state.

Note: **The Native API for Java transaction model**

The *Native API for Java* offers an alternative to the `java.sql` transaction model demonstrated here. The Native API transaction model is based on ObjectScript transaction methods, and is not interchangeable with the JDBC model. The Native API model must be used if your transactions include Native API method calls. See [Using the Native API for Java](#) for details.

5

JDBC Quick Reference

This chapter is a quick reference InterSystems extension methods in the following classes:

- [Class `ConnectionPoolDataSource`](#) — methods related to InterSystems connection pooling.
- [Class `IRISDataSource`](#) — InterSystems-specific connection properties.

Note: This chapter lists only the extension methods discussed in this book. See “[JDBC Driver Support](#)” in the *Implementation Reference for Java Third Party APIs* for a complete description of all InterSystems JDBC driver features, including extensions, variants, and implementation of optional JDBC features.

5.1 Class `ConnectionPoolDataSource`

The `com.intersystems.jdbc.ConnectionPoolDataSource` class fully implements the `javax.sql.ConnectionPoolDataSource` interface, and also includes the following extension methods to control InterSystems IRIS connection pooling:

`getMaxPoolSize()`

Extension method `ConnectionPoolDataSource.getMaxPoolSize()` returns an `int` representing the current maximum connection pool size.

```
int getMaxPoolSize()
```

`getPoolCount()`

Extension method `ConnectionPoolDataSource.getPoolCount()` returns an `int` representing the current number of entries in the connection pool.

```
int getPoolCount()
```

`getPooledConnection()`

Always use `IRISDataSource.getConnection()` instead of this method.

Do not call this method

`ConnectionPoolDataSource.getPooledConnection()` is required by the interface, but should never be invoked directly. The InterSystems JDBC driver controls connection pooling transparently.

restartConnectionPool()

Extension method `ConnectionPoolDataSource.restartConnectionPool()` restarts a connection pool. Closes all physical connections, and empties the connection pool.

```
void restartConnectionPool()
```

setMaxPoolSize()

Extension method `ConnectionPoolDataSource.setMaxPoolSize()` sets a maximum connection pool size. If the maximum size is not set, it defaults to 40.

```
void setMaxPoolSize(int max)
```

- `max` — optional maximum connection pool size (defaults to 40).

5.2 Class IRISDataSource

The `com.intersystems.jdbc.IRISDataSource` class fully implements the `javax.sql.DataSource` interface, and also includes numerous extension methods for getting or setting InterSystems IRIS connection properties (see “[Setting Connection Properties](#)” for more information).

`IRISDataSource` does not inherit the methods of `javax.sql.CommonDataSource`, which is not supported by the InterSystems JDBC driver.

getConnection()

Required Method with Extended Functionality

Required method `IRISDataSource.getConnection()` returns a `java.sql.Connection`. This method must always be used to obtain InterSystems IRIS driver connections. The InterSystems IRIS driver also provides pooling transparently through the `java.sql.Connection` object that `getConnection()` returns.

```
java.sql.Connection getConnection()  
java.sql.Connection getConnection(String usr,String pwd)
```

- `usr` — optional username argument for this connection.
- `pwd` — optional password argument for this connection.

This method provides pooling, and must always be used in place of `getPooledConnection()` and the methods of the `PooledConnection` class (see “[ConnectionPoolDataSource Extension Methods](#)” for more information).

getConnectionSecurityLevel()

Extension property accessor `IRISDataSource.getConnectionSecurityLevel()` returns an `int` representing the current Connection Security Level setting. Also see `setConnectionSecurityLevel()`.

```
int getConnectionSecurityLevel()
```

getDatabaseName()

Extension property accessor `IRISDataSource.getDatabaseName()` returns a `String` representing the current database (InterSystems IRIS namespace) name. Also see `setDatabaseName()`.

```
String getDatabaseName()
```

getDataSourceName()

Extension property accessor `IRISDataSource.getDataSourceName()` returns a `String` representing the current data source name. Also see [setDataSourceName\(\)](#).

```
String getDataSourceName()
```

getDefaultTransactionIsolation()

Extension property accessor `IRISDataSource.getDefaultTransactionIsolation()` returns an `int` representing the current default transaction isolation level. Also see [setDefaultTransactionIsolation\(\)](#).

```
int getDefaultTransactionIsolation()
```

getDescription()

Extension property accessor `IRISDataSource.getDescription()` returns a `String` representing the current description. Also see [setDescription\(\)](#).

```
String getDescription()
```

getEventClass()

Extension property accessor `IRISDataSource.getEventClass()` returns a `String` representing an Event Class object. Also see [setEventClass\(\)](#).

```
String getEventClass()
```

getKeyRecoveryPassword()

Extension property accessor `IRISDataSource.getKeyRecoveryPassword()` returns a `String` representing the current Key Recovery Password setting. Also see [setKeyRecoveryPassword\(\)](#).

```
String getKeyRecoveryPassword()
```

getNodeDelay()

Extension property accessor `IRISDataSource.getNodeDelay()` returns a `Boolean` representing a current `TCP_NODELAY` option setting. Also see [setNodeDelay\(\)](#).

```
boolean getNodeDelay()
```

getPassword()

Extension property accessor `IRISDataSource.getPassword()` returns a `String` representing the current password. Also see [setPassword\(\)](#).

```
String getPassword()
```

getPortNumber()

`IRISDataSource.getPortNumber()` returns an `int` representing the current port number. Also see [setPortNumber\(\)](#).

```
int getPortNumber()
```

getServerName()

Extension property accessor `IRISDataSource.getServerName()` returns a `String` representing the current server name. Also see [setServerName\(\)](#).

```
String getServerName()
```

getServicePrincipalName()

Extension property accessor `IRISDataSource.getServicePrincipalName()` returns a `String` representing the current Service Principal Name setting. Also see [setServicePrincipalName\(\)](#).

```
String getServicePrincipalName()
```

getSharedMemory()

Extension property accessor `IRISDataSource.getSharedMemory()` returns a `Boolean` indicating whether the connection is using shared memory. Also see [setSharedMemory\(\)](#).

```
Boolean getSharedMemory()
```

getSSLConfigurationName()

Extension property accessor `IRISDataSource.getSSLConfigurationName()` returns a `String` representing the current SSL Configuration Name setting. Also see [setSSLConfigurationName\(\)](#).

```
String getSSLConfigurationName()
```

getURL()

Extension property accessor `IRISDataSource.getURL()` returns a `String` representing a current URL for this data-source. Also see [setURL\(\)](#).

```
String getURL()
```

getUser()

Extension property accessor `IRISDataSource.getUser()` returns a `String` representing the current username. Also see [setUser\(\)](#).

```
String getUser()
```

setConnectionSecurityLevel()

Extension property accessor `IRISDataSource.setConnectionSecurityLevel()` sets the connection security level for this datasource. Also see [setConnectionSecurityLevel\(\)](#).

```
void setConnectionSecurityLevel(int level)
```

- `level` — connection security level number.

setDatabaseName()

Extension property accessor `IRISDataSource.setDatabaseName()` sets the database name (InterSystems IRIS namespace) for this datasource. Also see [getDatabaseName\(\)](#).

```
void setDatabaseName(String databaseName)
```

- `databaseName` — InterSystems IRIS namespace string.

setDataSourceName()

Extension property accessor `IRISDataSource.setDataSourceName()` sets the data source name for this datasource. `DataSourceName` is an optional setting and is not used to connect. Also see [getDataSourceName\(\)](#).

```
void setDataSourceName(String dataSourceName)
```

- `dataSourceName` — data source name string.

setDefaultTransactionIsolation()

Extension property accessor IRISDataSource.**setDefaultTransactionIsolation()** sets the default transaction isolation level. Also see [getDefaultTransactionIsolation\(\)](#).

```
void setDefaultTransactionIsolation(int level)
```

- `level` — default transaction isolation level number.

setDescription()

Extension property accessor IRISDataSource.**setDescription()** sets the description for this datasource. `Description` is an optional setting and is not used to connect. Also see [getDescription\(\)](#).

```
void setDescription(String desc)
```

- `desc` — datasource description string.

setEventClass()

Extension property accessor IRISDataSource.**setEventClass()** sets the Event Class for this datasource. The Event Class is a mechanism specific to InterSystems IRIS JDBC. It is completely optional, and the vast majority of applications will not need this feature. Also see [getEventClass\(\)](#).

```
void setEventClass(String eventClassName)
```

- `eventClassName` — name of transaction event class.

The InterSystems JDBC server will dispatch to methods implemented in a class when a transaction is about to be committed and when a transaction is about to be rolled back. The class in which these methods are implemented is referred to as the “event class.” If an event class is specified during login, then the JDBC server will dispatch to **%OnTranCommit** just prior to committing the current transaction and will dispatch to **%OnTranRollback** just prior to rolling back (aborting) the current transaction. User event classes should extend **%ServerEvent**. The methods do not return any values and cannot abort the current transaction.

setKeyRecoveryPassword()

Extension property accessor IRISDataSource.**setKeyRecoveryPassword()** sets the Key Recovery Password for this datasource. Also see [getKeyRecoveryPassword\(\)](#).

```
void setKeyRecoveryPassword(String password)
```

- `password` — datasource Key Recovery Password string.

setLogFile()

Extension property accessor IRISDataSource.**setLogFile()** unconditionally sets the log file name for this datasource.

```
void setLogFile(String logFile)
```

- `logFile` — datasource log file name string.

setNodelay()

Extension property accessor IRISDataSource.**setNodelay()** sets the `TCP_NODELAY` option for this datasource. Toggling this flag can affect the performance of the application. If not set, it defaults to `true`. Also see [getNodelay\(\)](#).

```
void setNodelay(boolean noDelay)
```

- `noDelay` — optional datasource TCP_NODELAY setting (defaults to `true`).

setPassword()

Extension property accessor `IRISDataSource.setPassword()` sets the password for this datasource. Also see [getPassword\(\)](#).

```
void setPassword(String pwd)
```

- `pwd` — datasource password string.

setPortNumber()

Extension property accessor `IRISDataSource.setPortNumber()` sets the port number for this datasource. Also see [getPortNumber\(\)](#).

```
void setPortNumber(int portNumber)
```

- `portNumber` — datasource port number.

setServerName()

Extension property accessor `IRISDataSource.setServerName()` sets the server name for this datasource. Also see [getServerName\(\)](#).

```
void setServerName(String serverName)
```

- `serverName` — datasource server name string.

setServicePrincipalName()

Extension property accessor `IRISDataSource.setServicePrincipalName()` sets the Service Principal Name for this datasource. Also see [getServicePrincipalName\(\)](#).

```
void setServicePrincipalName(String name)
```

- `name` — datasource Service Principal Name string.

setSharedMemory()

Extension property accessor `IRISDataSource.setSharedMemory()` sets shared memory connections for this datasource. Also see [getSharedMemory\(\)](#).

```
void setSharedMemory(Boolean sharedMemory)
```

setSSLConfigurationName()

Extension property accessor `IRISDataSource.setSSLConfigurationName()` sets the SSL Configuration Name for this datasource. Also see [getSSLConfigurationName\(\)](#).

```
void setSSLConfigurationName(String name)
```

- `name` — SSL Configuration Name string.

setURL()

Extension property accessor `IRISDataSource.setURL()` sets the URL for this datasource. Also see [getURL\(\)](#).

```
void setURL(String u)
```

- u — URL string.

setUser()

Extension property accessor `IRISDataSource.setUser()` sets the username for this datasource. Also see [getUser\(\)](#).

```
void setUser(String username)
```

- username — username string.

