Using Caché with JDBC

Version 2018.1
2024-11-07

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

Using Caché with JDBC
PDF generated on 2024-11-07
InterSystems Caché® Version 2018.1

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

About This Book

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

This book describes how to use the Caché JDBC driver, which enables you to connect to Caché from an external application (such as a development tool or report writer) via JDBC, and allows Caché to access external JDBC data sources.

To use the Caché 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 Caché JDBC driver on UNIX®, you should also be familiar with compiling and linking code, writing shell scripts, and other such tasks.

Organization of This Book

This book contains the following chapters:

- Introduction to Caché JDBC provides basic information about the Caché JDBC driver, and demonstrates how to set up and use a simple Caché JDBC application.
- Establishing JDBC Connections gives a detailed description of the various ways to establish a JDBC connection to a Caché database.
- Accessing JDBC Databases provides some simple examples that show how to access and manipulate data in a Caché database from a Java application using the Caché JDBC driver.
- Using the Caché SQL Gateway with JDBC describes how to access external databases from Caché using JDBC and the Caché SQL Gateway.
- Using the Caché Hibernate Dialect provides information on how to use Hibernate, an open source utility that acts as a wrapper around JDBC to provide object/relational mapping (ORM) services.
- Caché JDBC Compliance lists all members of the JDBC API, describes all Caché-specific additional features of the Caché JDBC driver, and indicates which optional features have been omitted.

Related Information

For related information, see the following sources:

- *Using Java with Caché XEP* describes the lightweight XEP event persistence technology for low latency object storage and processing.
- *Using Caché with ODBC* describes the Caché ODBC driver.

2

Introduction to Caché JDBC

Caché provides a high-performance *type 4* JDBC database driver. The Caché JDBC driver is fully compliant with the JDBC 4.1 API, supporting all required interfaces and adhering to all JDBC 4.1 guidelines and requirements.

The online *InterSystems Supported Platforms* document for this release specifies the current requirements for all Java-based binding applications:

- See "Supported Java Technologies" for supported Java releases.
- See "Supported Client Platforms" for supported JDBC client platforms.
- If your client application and the Caché server are not running on the same version of Caché, see "Supported Version Interoperability" for information on compatibility between versions.

You can use the Caché JDBC driver and SQL when you want to access your Caché data using a relational model. The Java Binding mechanism also uses the JDBC driver. You can mix relational and object-oriented database access to provide maximum flexibility to your application.

Using the JDBC Driver

The Caché JDBC driver is contained in cache-jdbc-2.0.0.jar. See "The Caché Java Class Packages" for information on using this file.

JavaDoc class documentation for the packages in cache-jdbc-2.0.0.jar is located in <install-dir>\dev\java\doc.

2.1 Installation and Configuration

All applications using the Caché Java binding are divided into two parts: a Caché server and a Java client. The Caché server is responsible for database operations as well as the execution of Caché object methods. The Java client is responsible for the execution of all Java code (such as additional business logic or the user interface). When an application runs, the Java client connects to and communicates with a Caché server via a TCP/IP socket. The actual deployment configuration is up to the application developer: the Java client and Caché server may reside on the same physical machine or they may be located on different machines. Only the Caché server machine requires a copy of Caché.

2.1.1 Java Client Requirements

The Caché Java client requires the Java JDK, version 1.7 or higher. Client applications do not require a local copy of Caché.

The online *InterSystems Supported Platforms* document for this release specifies the current requirements for all Java-based binding applications:

- See "Supported Java Technologies" for supported Java releases.
- See "Supported Client Platforms" for supported Java client platforms.
- If your client application and the Caché server are not running on the same version of Caché, see "Supported Version Interoperability" for information on compatibility between versions.

The core components of the Java binding are files named cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar, which contain the Java classes that provide the connection and caching mechanisms for communication with the Caché server, JDBC connectivity, and reflection support. Client applications do not require a local copy of Caché, but the cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar files must be on the class path of the application when compiling or using Java proxy classes. See "The Caché Java Class Packages" for more information on these files.

2.1.2 Caché Server Configuration

Very little configuration is required to use a Java client with a Caché server. The Java sample programs provided with Caché should work with no change following a default Caché installation. This section describes the server settings that are relevant to Java and how to change them.

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

The Java sample programs use the following connection information:

```
String url = "jdbc:Cache://127.0.0.1:1972/SAMPLES";
String user = "_SYSTEM";
String password = "SYS";
```

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 Caché server (see "Supported Version Interoperability" in the online *InterSystems Supported Platforms* document for this release. The client and the server can be running on the same machine.
- Your class path must include the versions of cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar that correspond to the client version of the Java JDK (see "The Caché Java Class Packages").
- To connect to the Caché server, the client application must have the following information:
  - The IP address of the machine on which the Caché 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 Caché SuperServer is listening. The Java sample programs use `1972` (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. The Java samples connect to the SAMPLES namespace, which is pre-installed with Caché.

  See "Establishing JDBC Connections" for detailed information on connecting to the Caché server.

2.2 The Caché Java Class Packages

The files containing the Caché Java class packages are located in <install-dir>\Dev\java\lib\<java-release>, where <install-dir> is the root directory of your Caché installation and <java-release> corresponds to the Java JDK you are using. See "Caché Installation Directory" in the *Caché Installation Guide* for the location of <install-dir> on your system. For supported Java releases, see "Supported Java Technologies" in the online *InterSystems Supported Platforms* document for this release.

The Java class packages are contained in the following files:

- cache-jdbc-2.0.0.jar — all of the other files in this list are dependent on this file. It can be used by itself for JDBC binding applications.
- cache-db-2.0.0.jar — required for most Java binding applications.
- cache-extreme-2.0.0.jar — required only for Caché XEP binding applications (see *Using Java with Caché XEP*).
- cache-gateway-2.0.0.jar — required for JDBC SQL Gateway applications (see "Using the Caché SQL Gateway with JDBC").

See the JavaDoc in <install-dir>\dev\java\doc\ for the latest and most complete description of these packages.

3

Establishing JDBC Connections

This chapter provides some detailed examples of Java code that uses the Caché JDBC driver to accomplish basic tasks.

- Using CacheDataSource to Connect — describes using CacheDataSource to load the driver and create a java.sql.Connection object, which is used to create the Statement object.
- Using DriverManager to Connect — describes using the standard DriverManager class to create a connection.
- Using a Connection Pool — describes using the CacheConnectionPoolDataSource class to control the connection pool for your Java client applications.
- Defining a JDBC Connection URL — describes how to specify the host address, port number, and namespace to be accessed. The Caché JDBC driver also allows you to specify several other parameters.
- Caché JDBC Connection Properties — describes how to specify several standard connection properties.

3.1 Using CacheDataSource to Connect

Use **com.intersys.jdbc.CacheDataSource** 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 Caché.

Note: In earlier versions of JDBC, it was necessary to load the driver before connecting. For example:

```
Class.forName ("com.intersys.jdbc.CacheDriver").newInstance();
Connection dbconnection = DriverManager.getConnection(url,user,password);
```

JDBC versions 4.0 and higher use driver autoloading, which makes the `Class.forName()` call unnecessary for either DriverManager or CacheDataSource.

Here are the steps for using CacheDataSource:

1. Import the java.sql and com.intersys.jdbc packages:

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

2. Load the driver, then use CacheDataSource to create the connection and specify username and password:

```
try{
   CacheDataSource ds = new CacheDataSource();
   ds.setURL("jdbc:Cache://127.0.0.1:1972/Samples");
   ds.setUser("_system");
   ds.setPassword("SYS");
   Connection dbconnection = ds.getConnection();
}
```

   Note: On some systems, Java may attempt to connect via IPv6 if you use `localhost` in the URL rather than the literal address, `127.0.0.1`. This applies on any system where the hostname resolves the same for IPv4 and IPv6.

3. Create a java.sql.Statement using the connection. The Statement object can be used to execute queries.

```
Statement stmt = dbconnection.createStatement()
```

4. Catch the exceptions thrown by the above methods:

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

3.2 Using DriverManager to Connect

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

```
Class.forName ("com.intersys.jdbc.CacheDriver").newInstance();
String  url="jdbc:Cache://127.0.0.1:1972/SAMPLES";
String  username = "_SYSTEM";
String  password = "SYS";
dbconnection = DriverManager.getConnection(url,username,password);
```

You can also pass connection properties to DriverManager in a Properties object, as demonstrated in the following code:

```
String  url="jdbc:Cache://127.0.0.1:1972/SAMPLES";
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 Caché JDBC Connection Properties for a complete list of the properties used by the Caché JDBC driver.

3.3 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 Caché JDBC driver also allows you to use several optional parameters.

3.3.1 Required Parameters

The URL specifies the host address, port number, and namespace to be accessed.

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 Caché SuperServer is listening. The default is `1972` (or the first available number higher than `56772` if more than one instance of Caché is installed — see DefaultPort in the *Caché Parameter File Reference*).
namespace	Caché namespace to be accessed. For example, `Samples` is the namespace containing Caché sample programs.

The minimal required URL syntax is:

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

For example, the following URL specifies *host* as `127.0.0.1`, *port* as `1972`, and *namespace* as `Samples`:

```
jdbc:Cache://127.0.0.1:1972/Samples
```

3.3.2 Optional Parameters

The Caché JDBC driver also allows you to specify several optional parameters. The full syntax is:

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

where the optional parameters are defined as follows:

logfile	specifies a JDBC log file (see "Enabling Logging for JDBC").
eventclass	sets the transaction Event Class for this **CacheDataSource** object. See the CacheDataSource **setEventClass()** method for a complete description.
nodelay	sets the `TCP_NODELAY` option if connecting via a **CacheDataSource** object. Toggling this flag can affect the performance of the application. Valid values are `true` and `false`. If not set, it defaults to `true`. Also see the **getNodelay()** and **setNodelay()** methods of CacheDataSource.
ssl	enables SSL/TLS for both CacheDriver and CacheDataSource (see "Using SSL/TLS with Caché" in the *Caché 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 setting the others. For example, the following URL sets only the required parameters and the *nodelay* option:

```
jdbc:Cache://127.0.0.1:1972/Samples/::false
```

3.3.3 Setting the Port Parameter at the Command Line

The com.intersys.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 1972 in a hard-coded connection string. The following command line will still allow it to run on port 9523:

```
java -cp .:../lib/cache-db-2.0.0.jar -Dcom.intersys.port=9523 myJdbcProgram
```

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

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

3.3.4 Alternate Username and Password Parameters

For the preferred ways to specify username and password, see "Using CacheDataSource to Connect" and "Using DriverManager to Connect" at the beginning of this chapter. 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:Cache://<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:Cache://127.0.0.1:1972/Samples/::false?username=_SYSTEM&password=SYS"
```

The username and password strings are case-sensitive.

3.4 Using a Connection Pool

The **com.intersys.jdbc.CacheConnectionPoolDataSource** 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 Caché:

1. Import the needed packages:

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

2. Instantiate a CacheConnectionPoolDataSource 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.

   ```
   CacheConnectionPoolDataSource pds = new CacheConnectionPoolDataSource();
      pds.restartConnectionPool();
      pds.setURL("jdbc:Cache://127.0.0.1:1972/Samples");
      pds.setUser("_system");
      pds.setPassword("SYS");
   ```

3. Initially, **getPoolCount** returns 0.

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

4. Use **CacheConnectionPoolDataSource.getConnection()** to retrieve a database connection from the pool.

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

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

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

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

3.5 Caché JDBC Connection Properties

The Caché JDBC driver supports several connection properties, which can be set by passing them to DriverManager (as described in Using DriverManager to Connect).

The following properties are supported:

`user`	Required. String indicating Username. Default = ""
`password`	Required. String indicating Password. Default = ""
`TCP_NODELAY`	Optional. Boolean indicating TCP/IP NoDelay Flag. Default = `true`.
`SO_SNDBUF`	Optional. Integer indicating TCP/IP `SO_SNDBUF` value (SendBufferSize). Default = `0` (use system default value).
`SO_RCVBUF`	Optional. Integer indicating TCP/IP `SO_RCVBUF` value(ReceiveBufferSize). Default = `0` (use system default value).
`TransactionIsolationLevel`	Optional. java.sql.Connection constant indicating Transaction Isolation Level. Valid values are `TRANSACTION_READ_UNCOMMITTED` (the default) or `TRANSACTION_READ_COMMITTED`.
`service principal`	Optional. String indicating Service Principal Name. Default = `null`.
`connection security level`	Optional. Integer indicating Connection Security Level. Valid levels are `0`, `1`, `2`, `3`, or `10`. Default = `0`. `0` - Caché login (Password) `1` - Kerberos (authentication only) `2` - Kerberos with Packet Integrity `3` - Kerberos with Encryption `10` - SSL/TLS

3.5.1 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\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]);
   }
}
```

4

Accessing JDBC Databases

This chapter provides examples of Java code using the Caché JDBC driver to query databases and work with the results.

- A Simple JDBC Application — a complete but very simple application that demonstrates the basic features of JDBC.
- Using Statements — an overview of JDBC SQL query classes.
- Inserting and Updating Data — using JDBC result sets to insert and update data in a Caché database.
- Logging for JDBC Applications — how to enable logging to test and troubleshoot your JDBC applications.

Note: In the examples given in this chapter, several methods throw exceptions of type SQLException. The required `try catch` blocks are omitted for clarity.

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:

- A CacheDataSource object is used to create a Connection object that links the JDBC application to the Caché 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 TinyJDBC Application

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

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

public class TinyJDBC{
  public static void main() {
    try {
```

Use CacheDataSource to open a connection (for details, see Using CacheDataSource to Connect):

```
Class.forName ("com.intersys.jdbc.CacheDriver").newInstance();
CacheDataSource ds = new CacheDataSource();
ds.setURL("jdbc:Cache://127.0.0.1:1972/SAMPLES");
Connection dbconn = ds.getConnection("_SYSTEM","SYS");
```

Execute a query and get a scrollable, updatable result set.

```
String sql="Select Name from Sample.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("TinyJDBC caught exception: "
             + ex.getClass().getName() + ": " + ex.getMessage());
    }
  } // end main()
} // end class TinyJDBC
```

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

4.2 Using Statements

The sql.java package provides three classes used to query databases and return a ResultSet:Statement, PreparedStatement, and CallableStatement. All three classes are instantiated by calls to Connection methods. The following sections discuss how to use these classes:

- Using Statement to Execute a SELECT — a very simple statement call.
- Executing a Prepared Statement — a more detailed example.
- Executing Stored Procedures with CallableStatement — an example that executes the **ByName** stored procedure from Sample.Person.
- Returning Multiple Result Sets — accessing multiple result sets returned by Caché stored procedures.
- Statement Pooling — how the Caché driver stores and uses optimized statements.

4.2.1 Using Statement to Execute a SELECT

The following code executes an SQL **SELECT** on Caché using the Statement class:

- Create a query string and execute it using the java.sql.Statement **execute()** method:

```
String stQuery="SELECT ID, Name from Sample.Person";
java.sql.ResultSet rs = stmt.executeQuery(stQuery);
```

  You should always use the fully qualified name java.sql.ResultSet to avoid clashes with com.intersys.classes.ResultSet

- Process and display the query results:

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

4.2.2 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 Sample.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 Sample.Employee.

The prepared statement is implemented just like a regular 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 Sample.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();
}
```

4.2.3 Using Callable Statements to Execute Stored Procedures

The following code executes **ByName**, a Caché stored procedure contained in Sample.Person:

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

```
String sql="call Sample.SP_Sample_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();
```

4.2.4 Returning Multiple Result Sets

Caché allows you to define a stored procedure that returns multiple result sets. The Caché JDBC driver supports the execution of such stored procedures. Here is an example of a Caché 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 ]
{
 $$$ResultSet("select Name from Sample.Person where Name %STARTSWITH :st")
 $$$ResultSet("select Name, DOB from Sample.Person where Name %STARTSWITH :st")
 Quit
}
```

Note: This stored procedure is not defined in Sample.Person. In order to try this example, you must first open Sample.Person in Atelier and add the class method shown above. The **$$$ResultSet** routine prepares and executes a SQL statement (which must be a string literal, available at compile time) and returns the resultset. To use it, you must add the declaration `include %occResultSet` at the start of the Sample.Person file (before the class definition, as shown here):

```
include %occResultSet
Class Sample.Person Extends (%Persistent, %Populate, %XML.Adaptor)
{ ...
```

Remember to recompile Sample.Person after making these changes.

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 Sample.Person_DRS(?)");
cs.setString(1,"A");
boolean success=cs.execute();
```

- Iterate through the pair of result sets displaying the data. Note that **getMoreResults** moves to the Statement object's next result set while **getResultSet** retrieves the current 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());
```

Note: By default **getMoreResults** closes the current result set before moving to the next. The Caché JDBC Driver does not support keeping the current result set open after moving to the next.

4.2.5 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.

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

The Caché JDBC implementation supports Statement methods **setPoolable()** and **isPoolable()** as hints to whether the statement in question should be pooled. Caché 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 ConnectionPoolDataSource. The optional javax.sql.StatementEventListener interface is unsupported (and irrelevant) for the same reason.

4.3 Inserting and Updating Data

There are several ways to insert and update Caché 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.
- Updating a Scrollable Result Set — accessing and changing the data in result set rows.
- Using Transactions — using the JDBC transaction API to commit or roll back changes.

The makeTestSSN() Method

In this section, several examples insert new rows into Sample.Person, which requires SSN (Social Security Number) as a unique key. The following method is used in these examples to generate a random SSN of the form nnn-nn-nnnn:

```
public static String makeTestSSN() {
  java.util.Random random = new java.util.Random();
  StringBuffer sb = new StringBuffer();
  for (int i=1; i<=9; i++) sb.append(random.nextInt(10));
  sb.insert(5,'-');
  sb.insert(3,'-');
  return sb.toString();
}
```

4.3.1 Inserting Data and Retrieving Generated Keys

The following code inserts a new row into Sample.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 Sample.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 (see The makeTestSSN() Method for an explanation of the **makeTestSSN()** call):

```
String SSN = makeTestSSN(); // 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 ID,Name,Age,SSN FROM Sample.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 Caché 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 ID, Name, SSN from Sample.Person "+
   " Where Name > ? Order By Name";
int scroll=ResultSet.TYPE_SCROLL_INSENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;

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

- 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();
```

4.3.3 Updating a Scrollable Result Set

The following code updates an open result set and saves the changes to the database:

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

```
String sql="Select Name, SSN from Sample.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 Caché ID column. ID values are defined automatically by Caché.

- 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 (see The makeTestSSN() Method for an explanation of the **makeTestSSN()** call). Finally, invoke **insertRow**:

```
rs.moveToInsertRow();
rs.updateString(1, "Abelson,Alan");
rs.updateString(2, makeTestSSN());
rs.insertRow();
```

4.3.4 Using Transactions

The Caché JDBC driver supports the JDBC transaction API.

- 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)
      }
   }
}
```

4.3.4.1 Transaction Handling Methods

Here is a brief summary of the java.sql.Connection methods used for transaction handling:

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.

setTransactionIsolation()

Sets the isolation level for a transaction. Caché supports the following JDBC transaction isolation levels:

- Connection.TRANSACTION_READ_UNCOMMITTED — Level 1. Permits dirty reads, non-repeatable reads, and phantom reads.
- Connection.TRANSACTION_READ_COMMITTED — Level 2. Prevents dirty reads, but allows non-repeatable and phantom reads.

getIsolationLevel()

Returns the current transaction isolation level for the Connection object.

4.4 Logging for JDBC Applications

If your applications encounter any problems, you can monitor by enabling the appropriate logging:

- Enabling Logging for JDBC
- Enabling Logging for the JDBC SQL Gateway

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.

When using the SQL Gateway with JDBC, you should be able to find out more about logging by consulting the documentation for the remote database to which you are connecting.

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

4.4.1 Enabling Logging for JDBC

To enable logging for JDBC when connecting to Caché, 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:Cache://127.0.0.1:1972/USER
```

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

```
jdbc:Cache://127.0.0.1:1972/USER/myjdbc.log
```

This log records the interaction from the perspective of the Caché 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:Cache://127.0.0.1:1972/USER/+myjdbc.log
```

See Defining a JDBC Connection URL for a complete list of connection parameters.

4.4.2 Enabling Logging for the JDBC SQL Gateway

The Caché SQL Gateway can also generate a log when used with JDBC. To enable this logging:

- In the Management Portal, go to System Administration > Configuration > Connectivity > JDBC Gateway Settings.
- Specify a value for **JDBC Gateway Log**. This should be the name of a log file (for example, jdbcSqlGateway.log) that will record the interaction between the gateway and the database.

5

Using the Caché SQL Gateway with JDBC

The Caché SQL Gateway allows Caché to access external databases via both JDBC and ODBC. This chapter contains the SQL Gateway information concerning JDBC connections. For a detailed description of the SQL Gateway, see the chapter on Using the Caché SQL Gateway in *Using Caché SQL*.

5.1 Creating JDBC SQL Gateway Connections for External Sources

This section describes how to create a JDBC logical connection definition for the SQL Gateway.

Caché maintains a list of SQL Gateway connection definitions, which are logical names for connections to external data sources. Each connection definition consists of a logical name (for use within Caché), information on connecting to the data source, and a username and password to use when establishing the connection. These connections are stored in the table %Library.sys_SQLConnection. You can export data from this table and import it into another Caché instance.

Note: **Controlling Gateway Logging and Java Version**

To monitor SQL Gateway problems, you can enable SQL Gateway logging (see "Enabling Logging for the SQL Gateway"). You can specify the version of Java to be used with the SQL Gateway by setting the Caché *JavaHome* parameter (see JavaHome in the *Caché Parameter File Reference*).

5.1.1 Creating a JDBC SQL Gateway Connection

To define a gateway connection for a JDBC-compliant data source, perform the following steps:

1. In the Management Portal, go to the System Administration > Connectivity > SQL Gateway Connections page.
2. Click **Create New Connection**.
3. On the **Gateway Connection** page, enter or choose values for the following fields:
   - For **Type**, choose **JDBC**.
   - **Connection Name** — Specify an identifier for the connection, for use within Caché.
   - **User** — Specify the name for the account to serve as the default for establishing connections, if needed.
   - **Password** — Specify the password associated with the default account.
   - **Driver name** — Full class name of the JDBC client driver.

- **URL** — Connection URL for the data source, in the format required by the JDBC client driver that you are using.
- **Class path** — Specifies a comma-separated list of additional JAR files to load.
- **Properties** — Optional string that specifies vendor-specific connection properties. If specified, this string should be of the following form:

  property=value;*property=value*;...

  See Caché JDBC Connection Properties for more information on connection properties.

For example, a typical connection might use the following values:

Setting	Value
Type	`JDBC`
Connection Name	`ConnectionJDBC1`
User	`JDBCUser`
Password	`JDBCPassword`
Driver name	`oracle.jdbc.driver.OracleDriver`
URL	`jdbc:oracle:thin:@//oraserver:1521/SID`
Class path	`/fill/path/to/ojdbc14.jar`
Properties	`oracle.jdbc.V8Compatibility=true; includeSynonyms=false;restrictGetTables=true`

For the other options, see "Implementation-specific Options."

4. Optionally test if the values are valid. To do so, click the **Test Connection** button. The screen will display a message indicating whether the values you have entered allow for a valid connection.
5. To create the named connection, click **Save**.
6. Click **Close**.

5.1.2 Creating a JDBC Connection to Caché via the SQL Gateway

Caché provides JDBC drivers and can be used as a JDBC data source. That is, a Caché instance can connect to itself or to another Caché instance via JDBC and the SQL Gateway. Specifically, the connection is from a namespace in one Caché to a namespace in the other Caché. To connect in this way, you need the same information that you need for any other external database: the connection details for the database driver that you want to use. This section provides the basic information.

5.1.2.1 Connecting to Caché as a JDBC Data Source

To configure a Caché instance (Caché-1) to use another Caché instance (Caché-2) as a JDBC data source, do the following:

1. Within Caché-1, use the SQL Gateway to create a JDBC connection to the namespace in Caché-2 that you want to use.
   - For **Type**, choose **JDBC**.
   - **Connection Name** — Specify an identifier for the connection, for use within Caché-1.
   - **User** — Specify the username needed to access Caché-2, if needed.

- **Password** — Specify the password for this user.
- **Driver name** — Use com.intersys.jdbc.CacheDriver
- **URL** — Connection URL for the data source, in the following format:

  ```
  jdbc:Cache://IP_address:port/namespace
  ```

  Here *IP_address:port* is the IP address and TCP port where Caché-2 is running, and *namespace* is the namespace to which you want to connect (see Defining a JDBC Connection URL).

For example, a typical connection might use the following values:

Setting	Value
Type	`JDBC`
Connection Name	`Cache2Samples`
User	`_SYSTEM`
Password	`SYS`
Driver name	`com.intersys.jdbc.CacheDriver`
URL	`jdbc:Cache://127.0.0.1:1972/SAMPLES`

- **Class path** — Leave this blank.
- **Properties** — Optional string that specifies connection properties supported by the Caché JDBC drivers. If specified, this string should be of the following form:

  property=value;*property=value*;...

- **Conversion in composite Row IDs** — The default option should be suitable in most cases, but you can choose any option. The details are given in "Implementation-specific Options."
- **Do not use delimited identifiers by default** — The default (not to click this check box) should be suitable in most cases, but you can enable this option if desired. The details are given in "Implementation-specific Options."

2. Click **Save**.
3. Click **Close**.

5.1.3 Implementation-specific Options

Before you define an SQL gateway connection, you should make sure that you understand the requirements of the external database and of the database driver, because these requirements affect how you define the connection.

Do Not Use Delimited Identifiers by Default

The **Do not use delimited identifiers by default** option controls the format of identifiers in the generated routines.

Select this check box if you are using a database that does not support delimited SQL identifiers. This currently includes the following databases:

- Sybase
- Informix
- MS SQL Server

Clear the check box if you are using any other database. All SQL identifiers will be delimited.

Use COALESCE

The **Use COALESCE** option controls how a query is handled when it includes a parameter (?), and it has an effect only when a query parameter equals null.

- If you do not select **Use COALESCE** and if a query parameter equals null, the query returns only records that have null for the corresponding value. For example, consider a query of the following form:

  ```
  SELECT ID, Name from LinkedTables.Table WHERE Name %STARTSWITH ?
  ```

  If the provided parameter is null, the query would return only rows with null-valued names.

- If you select **Use COALESCE**, the query wraps each parameter within a `COALESCE` function call, which controls how null values are handled.

  Then, if a query parameter equals null, the query essentially treats the parameter as a wildcard. In the previous example, if the provided parameter is null, this query returns all rows.

Whether you select this option depends on your preferences and on whether the external database supports the `COALESCE` function.

To find out whether the external database supports the `COALESCE` function, consult the documentation for that database.

Conversion in Composite Row IDs

The **Conversion in composite Row IDs** option controls how non-character values are treated when forming a composite ID. Choose an option that is supported by your database:

- **Do not convert non-character values** — This option performs no conversion. This option is suitable only if your database supports concatenating non-character values to character values.
- **Use CAST** — This option uses `CAST` to convert non-character values to character values.
- **Use {fn convert ...}** — This option uses `{fn convert ...}` to convert non-character values to character values.

In all cases, the IDs are concatenated with `||` between the IDs (or transformed IDs).

Consult the documentation for the external database to find out which option or options it supports.

6

Using the Caché Hibernate Dialect

Java Persistence Architecture (JPA) is the recommended persistence technology for complex object hierarchies in Java projects. Caché currently supports JPA 1.0 and 2.0 via the Hibernate implementations of the JPA specifications.

Hibernate is an open source framework from JBoss that acts as a wrapper around JDBC to provide object/relational mapping (ORM) services for relational databases. Hibernate provides a vendor-neutral persistence service, which may be a requirement for some projects.

Since every vendor's implementation of SQL is slightly different, Hibernate includes vendor-provided "dialects" that customize its mappings to specific databases. Current Hibernate distributions (3.2.1GA and higher) include a high performance, customized Caché dialect class.

6.1 Requirements

This document assumes that the following software is installed on your system:

- Caché 2010.2 or higher
- Hibernate 3.2.1GA or higher (any version that includes the Caché dialect extensions). Hibernate can be downloaded from www.hibernate.org.
- A supported version of the Java JDK (see "Supported Java Technologies" in the online *InterSystems Supported Platforms* document for this release).

6.2 Installation and Configuration

This section provides instructions for setting up your system to use Hibernate with Caché. The instructions assume that the correct versions of both Caché and Hibernate are installed and operational.

6.2.1 Directories

The instructions in this chapter refer to the following directories:

- <install-dir> — the Caché installation directory (for the location of <install-dir> on your system, see "Default Caché Installation Directory" in the *Caché Installation Guide*).
- <hibernate_root> — the Hibernate installation directory (C:\hibernate-3.2 by default).

6.2.2 System Settings

Make the following changes to your system:

- *cache-jdbc-2.0.0.jar File*

  The cache-jdbc-2.0.0.jar file contains the Caché JDBC driver. If you haven't already done so, copy the appropriate version of cache-jdbc-2.0.0.jar (see "The Caché Java Class Packages") from the default location to:

  <hibernate_root>\lib\cache-jdbc-2.0.0.jar.

- *Java Classpath*

  Make sure the following items are on your Java classpath:

  – The jar files from <hibernate_root>\lib

  – The directory or directories where the Hibernate configuration files (hibernate.properties and hibernate.cfg.xml) are kept. By default, both files are in:

    <hibernate_root>\etc

6.2.3 Hibernate Configuration

In the Hibernate configuration files (either hibernate.properties or hibernate.cfg.xml), specify the connection information for your database, and the name of the Caché dialect class.

The following five configuration properties are required:

- *dialect* — The fully qualified name of the Caché dialect class. The base dialect class is:

  ```
  org.hibernate.dialect.Cache71Dialect
  ```

  You can use a custom dialect class derived from this base class if you need to enable support for the Hibernate primary key generator classes (see "Support for Sequences").

- *driver_class* — The fully qualified name of the Caché JDBC driver:

  ```
  com.intersys.jdbc.CacheDriver
  ```

  The JDBC driver is contained in the cache-jdbc-2.0.0.jar file (see "System Settings" for details).

- *username* — Username for the Caché namespace you want to access (default is _SYSTEM).

- *password* — Password for the Caché namespace (default is SYS).

- *url* — The URL for the Caché JDBC driver. The format for the URL is:

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

  where <host> is the IP address of the machine hosting Caché, <port> is the SuperServer TCP port of your Caché instance, and <namespace> is the namespace that contains your Caché database data (see Defining a JDBC Connection URL for more details).

A typical entry in hibernate.properties would contain the following lines:

```
hibernate.dialect   org.hibernate.dialect.Cache71Dialect
hibernate.connection.driver_class   com.intersys.jdbc.CacheDriver
hibernate.connection.username   _SYSTEM
hibernate.connection.password   SYS
hibernate.connection.url   jdbc:Cache://127.0.0.1:1972/USER
```

The following example shows the same information as it would appear in hibernate.cfg.xml:

```
<hibernate-configuration>
  <session-factory>
    <property name="dialect">
      org.hibernate.dialect.Cache71Dialect
    </property>
    <property name="connection.driver_class">
      com.intersys.jdbc.CacheDriver</property>
    <property name="connection.username">_SYSTEM</property>
    <property name="connection.password">SYS</property>
    <property name="connection.url">
      jdbc:Cache://127.0.0.1:1972/USER
    </property>
  </session-factory>
</hibernate-configuration>
```

CAUTION: If the same property is set in both hibernate.properties and hibernate.cfg.xml, Hibernate will use the value from hibernate.cfg.xml.

6.3 Supported Features

6.3.1 Using Hibernate with SQL

The following SQL features are not supported by the Caché Hibernate dialect:

- The `longvarbinary` and `longvarchar` types are not supported in a `WHERE` clause.
- Row valued expressions such as `WHERE (A,B,C)=(1,2,3)` are not supported.

6.3.2 Support for Sequences

The base class for Caché dialects is Cache71Dialect in package org.hibernate.dialect. You can extend this class to enable support for the optional Hibernate primary key generator methods. This support is required if you want to enable Oracle-style sequences or use other non-standard generation methods.

Adding the Extended Class

Create the following class:

```
public Cache71SequenceDialect extends Cache71Dialect {
   public boolean supportsSequences() {
      return true;
   }
}
```

Specify the fully qualified name of your new class in the dialect property of your hibernate.properties or hibernate.cfg.xml file (see the examples in "Hibernate Configuration"). For example, if your package was named mySeqDialect, the entry in hibernate.properties would be:

```
hibernate.dialect  mySeqDialect.Cache71SequenceDialect
```

In hibernate.cfg.xml, the entry would be:

```
<property name="dialect">
   mySeqDialect.Cache71SequenceDialect
</property>
```

Installing and Using Sequence Dialect Support

To install Hibernate sequence dialect support, load the CacheSequences project into the namespace that your application will access:

1. In Studio, change to the namespace your application will access.
2. From the Studio menu, select `Tools > Import Local...`
3. In the file dialog, open the CacheSequences.xml project file:

   <hibernate_root>/etc/CacheSequences.xml.

   Studio will load and compile the Caché InterSystems.Sequences class.

The sequence generator to be used is specified in the `id` property of your Hibernate mapping. Oracle-style sequences can be generated by specifying `sequence` or `seqhilo`. If `native` is specified, the `identity` generator will be used when Caché is the underlying database. For more information on these classes, see the Hibernate reference documentation (specifically, the id section in chapter 5, *Basic O/R Mapping*).

The following example demonstrates the use of a sequence generator in a Hibernate mapping:

```
<id name="id" column="uid" type="long" unsaved-value="null">
   <generator class="seqhilo">
      <param name="sequence">EVENTS_SEQ</param>
      <param name="max_lo">0</param>
   </generator>
</id>
```

The sequence classes maintain a table of counters that can be incremented by calling the stored procedure just before performing the insert. For example:

```
call InterSystems.Sequences_GetNext("Name")
// perform the insert here
```

You can also increment the table with a standard SQL statement. For example:

```
SELECT InterSystems.Sequences_GetNext(sequencename)
   FROM InterSystems.Sequences
   WHERE Name='sequencename'
```

The table can be queried as table `InterSystems.Sequences`. The data is actually stored in global `^InterSystems.Sequences`. You can make the sequences system-wide by mapping `^InterSystems.Sequences*` to a location that is common to all your applications.

7

Caché JDBC Compliance

The InterSystems JDBC Driver is a fully compliant type 4 implementation of the JDBC 4.2 standard. This chapter lists all classes and interfaces of the JDBC 4.2 API, indicates the level of support for each one, and describes all InterSystems-specific features. The following topics are discussed:

- JDBC and the InterSystems JDBC Driver — provides an overview and resource links for the JDBC driver.
- JDBC Driver Compliance — lists all classes and interfaces specified by the JDBC standard, and indicates the current level of support.
- Variants and Unsupported Optional Methods — provides details on classes that include permitted variances from the standard.
- InterSystems Enhancements and Extensions — lists and discusses InterSystems extensions to the standard JDBC API.

7.1 JDBC and the InterSystems JDBC Driver

The Java JDBC API is the industry standard for vendor-neutral database connectivity. It provides a reliable way for Java applications to connect to data sources on any supported platform and to query or perform operations on them with SQL.

InterSystems JDBC is implemented in a type 4 driver to deliver the highest possible performance. *Type 4* means that it is a direct-to-database pure Java driver, installed inside the client JVM and requiring no external software support. It is fully compliant with the JDBC 4.2 API specification, supporting all required interfaces and adhering to all JDBC 4.2 guidelines and requirements. InterSystems JDBC supports all features except SQL Exception handling enhancements, National Character Set conversions, and the XML data type.

7.1.1 Installation and Configuration

The InterSystems JDBC driver is included in the standard InterSystems installation package. No extra installation or setup procedures are required. See "Installation and Configuration" for information on client requirements and usage.

7.2 JDBC Driver Compliance

This section provides information on the level of support for each JDBC interface.

7.2.1 Required java.sql Interfaces

The following interfaces must be implemented. Some classes contain methods that are optional if the implementation depends on a feature that the database does not support. The *standard implementation* annotation indicates that the generic implementation of the class has been used without alteration:

- java.sql.CallableStatement — implemented with some permitted variances (see "CallableStatement: Unsupported Methods" and "CallableStatement getBinaryStream() Extension Method").
- java.sql.ClientInfoStatus — *standard implementation*.
- java.sql.Connection — implemented with some permitted variances (see Connection: Unsupported or Restricted Methods).
- java.sql.DatabaseMetaData — implemented with some permitted variances (see "DatabaseMetaData: Variant Methods").
- java.sql.Date — *standard implementation*.
- java.sql.Driver — implemented with some permitted variances (see "Driver: Unsupported Methods").
- java.sql.DriverManager — *standard implementation*.
- java.sql.DriverPropertyInfo — *standard implementation*.
- java.sql.ParameterMetaData — all methods fully supported.
- java.sql.PreparedStatement — implemented with some permitted variances (see "PreparedStatement: Unsupported Methods").
- java.sql.ResultSet — implemented with some permitted variances (see "ResultSet: Unsupported or Restricted Methods") .
- java.sql.ResultSetMetaData — all methods fully supported.
- java.sql.RowIdLifeTime — *standard implementation*.
- java.sql.SQLPermission — *standard implementation*.
- java.sql.Statement — implemented with some permitted variances (see "Statement: Unsupported or Restricted Methods").
- java.sql.Time — *standard implementation*.
- java.sql.Timestamp — *standard implementation*.
- java.sql.Types — *standard implementation*.
- java.sql.Wrapper — all methods fully supported.

7.2.2 Optional java.sql Interfaces

All optional java.sql interfaces are listed below. *Italicized* items are not implemented:

- *java.sql.Array*
- java.sql.Blob — all methods fully supported.
- java.sql.Clob — all methods fully supported.
- java.sql.NClob — all methods fully supported.
- *java.sql.Ref*
- java.sql.RowId — all methods fully supported.

- java.sql.Savepoint — all methods fully supported.
- *java.sql.SQLData*
- *java.sql.SQLInput*
- *java.sql.SQLOutput*
- *java.sql.SQLXML*
- *java.sql.Struct*

7.2.3 java.sql Exceptions

The InterSystems JDBC driver throws only the following exceptions:

- java.sql.BatchUpdateException
- java.sql.SQLException
- java.sql.SQLWarning

The following exceptions are listed here for completeness, but are not required and are never used:

- *DataTruncation*
- *SQLClientInfoException*
- *SQLDataException*
- *SQLFeatureNotSupportedException*
- *SQLIntegrityConstraintViolationException*
- *SQLInvalidAuthorizationSpecException*
- *SQLNonTransientConnectionException*
- *SQLNonTransientException*
- *SQLRecoverableException*
- *SQLSyntaxErrorException*
- *SQLTimeoutException*
- *SQLTransactionRollbackException*
- *SQLTransientConnectionException*
- *SQLTransientException*

7.2.4 Required javax.sql Interfaces

The following required interfaces are supported. The *standard implementation* annotation indicates that the generic implementation of the class has been used without alteration:

- javax.sql.ConnectionEvent — *standard implementation*.
- javax.sql.DataSource — implemented with enhancements and additional methods (see "DataSource Extensions and Enhancements" for details).
- javax.sql.RowSetEvent — *standard implementation*.
- javax.sql.StatementEvent — *standard implementation*.

7.2.5 Optional javax.sql Interfaces

All optional javax.sql interfaces are listed below. *Italicized* items are not implemented:

- *javax.sql.CommonDataSource* — not implemented. Use javax.sql.DataSource instead (see "DataSource Extensions and Enhancements" for related information).
- javax.sql.ConnectionEventListener — all methods fully supported.
- javax.sql.ConnectionPoolDataSource — implemented with variants and additional methods (see "ConnectionPoolDataSource Extensions and Enhancements" for details).
- javax.sql.PooledConnection — all methods fully supported.
- *javax.sql.Rowset*
- *javax.sql.RowSetInternal*
- *javax.sql.RowSetListener*
- *javax.sql.RowSetMetaData*
- *javax.sql.RowSetReader*
- *javax.sql.RowSetWriter*
- *javax.sql.StatementEventListener*
- *javax.sql.XAConnection*
- *javax.sql.XADataSource*

7.3 Variants and Unsupported Optional Methods

The following interfaces have optional methods that the InterSystems JDBC driver does not support, or methods implemented in a non-standard manner:

- CallableStatement: Unsupported Methods
- Connection: Unsupported or Restricted Methods
- DatabaseMetaData: Variant Methods
- Driver: Unsupported Methods
- PreparedStatement: Unsupported Methods
- ResultSet: Unsupported or Restricted Methods
- Statement: Unsupported or Restricted Methods

7.3.1 CallableStatement: Unsupported Methods

Unsupported Optional Methods

java.sql.CallableStatement does not support the following optional methods:

- **getArray()**

```
Array getArray(int i)
Array getArray(String parameterName)
```

- **getObject()**

```
Object getObject(int i, java.util.Map map)
Object getObject(String parameterName, java.util.Map map)
```

- **getRef()**

```
Ref getRef(int i)
Ref getRef(String parameterName)
```

- **getRowId()** and **setRowId()**

```
java.sql.RowId getRowId(int i)
java.sql.RowId getRowId(String parameterName)

void setRowId(String parameterName, java.sql.RowId x)
```

- **getURL()** and **setURL()**

```
java.net.URL getURL(int i)
java.net.URL getURL(String parameterName)

void setURL(String parameterName, java.net.URL val)
```

- **getSQLXML()** and **setSQLXML()**

```
java.sql.SQLXML getSQLXML(int parameterIndex)
java.sql.SQLXML getSQLXML(String parameterName)

void setSQLXML(String parameterName, java.sql.SQLXML xmlObject)
```

Note: The java.sql.CallableStatement class also has one InterSystems extension method, which is discussed elsewhere (see "CallableStatement getBinaryStream() Extension Method").

7.3.2 Connection: Unsupported or Restricted Methods

Unsupported Optional Methods

java.sql.Connection does not support the following optional methods:

- **abort()**

```
void abort(Executor executor)
```

- **createArrayOf()**

```
java.sql.Array createArrayOf(String typeName, Object[] elements)
```

- **createBlob()**

```
Blob createBlob()
```

- **createClob()**

```
Clob createClob()
```

- **createNClob()**

```
java.sql.NClob createNClob()
```

- **createSQLXML**()

```
java.sql.SQLXML createSQLXML()
```

- **createStruct()**

```
java.sql.Struct createStruct(String typeName, Object[] attributes)
```

- **getTypeMap()**

```
java.util.Map getTypeMap()
```

- **setTypeMap()**

```
void setTypeMap(java.util.Map map)
```

Optional Methods with Restrictions

The following optional java.sql.Connection methods are implemented with restrictions or limitations:

- **prepareCall**()

  Only `TYPE_FORWARD_ONLY` is supported for *resultSetType*. Only `CONCUR_READ_ONLY` is supported for *resultSetConcurrency*.

```
java.sql.CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)
```

- **setReadOnly**()

  A no-op (the InterSystems JDBC driver does not support `READ_ONLY` mode)

```
void setReadOnly(Boolean readOnly)
```

- **setCatalog**()

  A no-op (the InterSystems JDBC driver does not support catalogs)

```
void setCatalog(String catalog)
```

- **setTransactionIsolation**()

  Only `TRANSACTION_READ_COMMITED` and `TRANSACTION_READ_UNCOMMITED` are supported for *level*.

```
void setTransactionIsolation(int level)
```

The following java.sql.Connection methods do not support `CLOSE_CURSORS_AT_COMMIT` for *resultSetHoldability*:

- **createStatement()**

```
java.sql.Statement createStatement(int resultSetType, int result, int resultSetHoldability)
```

- **prepareCall**()

```
java.sql.CallableStatement prepareCall(String sql,
                                       int resultSetType,
                                       int resultSetConcurrency,
                                       int resultSetHoldability)
```

- **prepareStatement**()

```
java.sql.PreparedStatement prepareStatement(String sql,
                                            int resultSetType,
                                            int resultSetConcurrency,
                                            int resultSetHoldability)
```

InterSystems JDBC currently supports only zero or one Auto Generated Keys. An exception is thrown if the java.sql.Connection methods below provide *columnIndexes* or *columnNames* arrays whose lengths are not equal to one.

- **prepareStatement**()

```
java.sql.PreparedStatement prepareStatement(String sql, int[] columnIndexes)
java.sql.PreparedStatement prepareStatement(String sql, String[] columnNames)
```

7.3.3 DatabaseMetaData: Variant Methods

Variant Methods

java.sql.DatabaseMetaData is fully supported, but has methods that vary from the JDBC standard due to InterSystems-specific handling of their return values. The following methods are affected:

- **supportsMixedCaseQuotedIdentifiers()**

  InterSystems JDBC returns `false`, which is not JDBC compliant.

```
boolean supportsMixedCaseQuotedIdentifiers()
```

- **getIdentifierQuoteString()**

  If delimited id support is turned on, InterSystems JDBC returns " (double quote character), which is what a JDBC compliant driver should return; otherwise InterSystems JDBC returns a space.

```
String getIdentifierQuoteString()
```

7.3.4 Driver: Unsupported Methods

Unsupported Optional Method

java.sql.Driver does not support the following optional method:

- **getParentLogger()**

```
void getParentLogger()
```

7.3.5 PreparedStatement: Unsupported Methods

Unsupported Optional Methods

java.sql.PreparedStatement does not support the following optional methods:

- **setArray()**

```
void setArray(int i, Array x)
```

- **setRef()**

```
void setRef(int i, Ref x)
```

- **setRowId()**

```
void setRowId(int parameterIndex, RowId x)
```

- **setSQLXML()**

```
void setSQLXML(int parameterIndex, SQLXML xmlObject)
```

- **setUnicodeStream()**

  Deprecated in Java JDK specification.

```
void setUnicodeStream(int i, InputStream x, int length)
```

- **setURL()**

```
void setURL(int i, java.net.URL x)
```

7.3.6 ResultSet: Unsupported or Restricted Methods

Optional Method with Restrictions

InterSystems JDBC does not support `TYPE_SCROLL_SENSITIVE` result set types. The following method is implemented with restrictions:

- **setFetchDirection()**

  Does not support `ResultSet.FETCH_REVERSE` (instead, use **afterLast** to move the result set's cursor to after the last row, and use **previous** to scroll backwards).

```
void setFetchDirection(int direction)
```

Unsupported Optional Methods

java.sql.ResultSet does not support the following optional methods:

- **getArray()**

```
Array getArray(int i)
Array getArray(String colName)
```

- **getCursorName()**

```
String getCursorName()
```

- **getObject()**

```
Object getObject(int i, java.util.Map map)
Object getObject(String colName, java.util.Map map)
```

- **getRef()**

```
Ref getRef(int i)
Ref getRef(String colName)
```

- **getHoldability()**

```
int getHoldability()
```

- **getUnicodeStream()**

Deprecated in Java JDK specification.

```
java.io.InputStream getUnicodeStream(int i)
java.io.InputStream getUnicodeStream(String colName)
```

- **getURL()**

```
java.net.URL getURL(int i)
java.net.URL getURL(String colName)
```

- **updateArray()**

```
void updateArray(int i, Array x)
void updateArray(String colName, Array x)
```

- **updateRef()**

```
void updateRef(int i, Ref x)
void updateRef(String colName, Ref x)
```

7.3.7 Statement: Unsupported or Restricted Methods

Unsupported Optional Methods

java.sql.Statement does not support the following optional methods:

- **cancel()**

```
void cancel()
```

- **closeOnCompletion()**

```
void closeOnCompletion()
```

- **isCloseOnCompletion()**

```
boolean isCloseOnCompletion()
```

Optional Methods with Restrictions

The following optional java.sql.Statement methods are implemented with restrictions or limitations:

- **getResultSetHoldability()**

  Only `HOLD_CURSORS_OVER_COMMIT`

```
int getResultSetHoldability()
```

- **setCursorName()**

  A no-op.

```
void setCursorName(String name)
```

- **setEscapeProcessing()**

  A no-op (does not apply)

```
void setEscapeProcessing(Boolean enable)
```

- **setFetchDirection()**

Does not support `ResultSet.FETCH_REVERSE` (instead, use **afterLast** to move the result set's cursor to after the last row, and use **previous** to scroll backwards).

```
void setFetchDirection(int direction)
```

InterSystems JDBC currently supports only zero or one auto-generated key. An exception is thrown if the java.sql.Statement methods below provide *columnIndexes* or *columnNames* arrays whose lengths are not equal to one:

- **execute()**

```
boolean execute(String sql, int[] columnIndexes)
boolean execute(String sql, String[] columnNames)
```

- **executeUpdate()**

```
int executeUpdate(String sql, int[] columnIndexes)
int executeUpdate(String sql, String[] columnNames)
```

7.4 InterSystems Enhancements and Extensions

The following classes provide additional InterSystems-specific extension methods:

- CallableStatement getBinaryStream() Extension Method
- ConnectionPoolDataSource Extensions and Enhancements discusses com.intersys.jdbc.CacheConnectionPoolDataSource, which is the InterSystems implementation of the javax.sql.ConnectionPoolDataSource interface.
- DataSource Extensions and Enhancements discusses com.intersys.jdbc.CacheDataSource, which is the InterSystems implementation of javax.sql.DataSource.

7.4.1 CallableStatement getBinaryStream() Extension Method

java.sql.CallableStatement implements the following additional InterSystems-specific extension method:

- **getBinaryStream()**

  Retrieves the value of the designated parameter (where *i* is the index of the parameter) as a java.io.InputStream object.

  ```
  java.io.InputStream getBinaryStream(int i)
  ```

  This method is a complement to the standard **setBinaryStream()** method, and an alternative to **getCharacterStream()** (which returns java.io.Reader).

7.4.2 ConnectionPoolDataSource Extensions and Enhancements

The com.intersys.jdbc.CacheConnectionPoolDataSource class fully implements the javax.sql.ConnectionPoolDataSource interface. This class does not inherit the methods of javax.sql.CommonDataSource, which is not supported by the InterSystems JDBC driver.

Restricted Method

getPooledConnection() is implemented because it is required by the JDBC standard, but the InterSystems implementation should never be called directly. InterSystems JDBC driver connections must always be obtained by calling the **getConnection()** method. (See "Using a Connection Pool" for more information).

- **getPooledConnection()**

```
javax.sql.PooledConnection getPooledConnection()
javax.sql.PooledConnection getPooledConnection(String usr,String pwd)
```

CAUTION: Calling applications should never use the **getPooledConnection()** methods or the PooledConnection class. InterSystems JDBC driver connections must always be obtained by calling the **getConnection()** method (which is inherited from CacheDataSource). The InterSystems JDBC driver provides pooling transparently through the java.sql.Connection object that it returns.

CacheConnectionPoolDataSource inherits from CacheDataSource (see "DataSource Extensions and Enhancements"), which provides additional InterSystems extension methods.

7.4.2.1 ConnectionPoolDataSource Extension Methods

CacheConnectionPoolDataSource also supports the following additional InterSystems-only management methods (see "Using a Connection Pool" for more information).

- **restartConnectionPool()**

  Restarts a connection pool. Closes all physical connections, and empties the connection pool.

```
void restartConnectionPool()
```

- **getPoolCount()**

  Returns the current number of entries in the connection pool.

```
int getPoolCount()
```

- **setMaxPoolSize()**

  Sets a maximum connection pool size. If the maximum size is not set, it defaults to 40.

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

- **getMaxPoolSize()**

  Returns the current maximum connection pool size

```
int getMaxPoolSize()
```

7.4.3 DataSource Extensions and Enhancements

The com.intersys.jdbc.CacheDataSource class fully implements the javax.sql.DataSource interface. This class does not inherit the methods of javax.sql.CommonDataSource, which is not supported by the InterSystems JDBC driver.

Enhanced Required Method

The InterSystems JDBC implementation of this method is enhanced to provide automatic, transparent connection pooling (see "Using a Connection Pool" for more information).

- **getConnection()**

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

7.4.3.1 DataSource Extension Methods

In addition to the methods defined by the interface, CacheDataSource also implements the following methods that can be used to get or set DataSource properties supported by InterSystems JDBC. (See "Installation and Configuration" for more information).

- **getConnectionSecurityLevel()**

  Returns an *int* representing the current Connection Security Level setting.

  ```
  int getConnectionSecurityLevel()
  ```

- **getDatabaseName()**

  Returns a String representing the current database (Caché namespace) name.

  ```
  String getDatabaseName()
  ```

- **getDataSourceName()**

  Returns a String representing the current data source name.

  ```
  String getDataSourceName()
  ```

- **getDefaultTransactionIsolation()**

  Gets the current default transaction isolation.

  ```
  int getDefaultTransactionIsolation()
  ```

- **getDescription()**

  Returns a String representing the current description.

  ```
  String getDescription()
  ```

- **getEventClass()**

  Returns a String representing an Event Class object.

  ```
  String getEventClass()
  ```

- **getKeyRecoveryPassword()**

  Returns a String representing the current Key Recovery Password setting.

  ```
  getKeyRecoveryPassword()
  ```

- **getNodelay()**

  Returns a boolean representing a current TCP_NODELAY option setting.

  ```
  boolean getNodelay()
  ```

- **getPassword()**

  Returns a String representing the current password.

  ```
  String getPassword()
  ```

- **getPortNumber()**

Returns an *int* representing the current port number.

```
int getPortNumber()
```

- **getServerName()**

  Returns a String representing the current server name.

  ```
  String getServerName()
  ```

- **getServicePrincipalName()**

  Returns a String representing the current Service Principal Name setting.

  ```
  String getServicePrincipalName()
  ```

- **getSSLConfigurationName()**

  Returns a String representing the current SSL Configuration Name setting.

  ```
  getSSLConfigurationName()
  ```

- **getURL()**

  Returns a String representing a current URL for this connection.

  ```
  String getURL()
  ```

- **getUser()**

  Returns a String representing the current username.

  ```
  String getUser()
  ```

- **setConnectionSecurityLevel()**

  Sets the connection security level

  ```
  Sets the Connection Security Level for this DataSource object.
  ```

- **setDatabaseName()**

  Sets the database name (Caché namespace) for this connection.

  ```
  void setDatabaseName(String dn)
  ```

- **setDataSourceName()**

  Sets the data source name for this connection. `DataSourceName` is an optional setting and is not used by CacheDataSource to connect.

  ```
  void setDataSourceName(String dsn)
  ```

- **setDefaultTransactionIsolation()**

  Sets the default transaction isolation level.

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

- **setDescription()**

Sets the description for this connection. `Description` is an optional setting and is not used by CacheDataSource to connect.

```
void setDescription(String d)
```

- **setEventClass()**

  Sets the Event Class for this connection. The Event Class is a mechanism specific to InterSystems JDBC. It is completely optional, and the vast majority of applications will not need this feature.

  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.

  ```
  void setEventClass(String e)
  ```

- **setKeyRecoveryPassword()**

  Sets the Key Recovery Password for this connection.

  ```
  setKeyRecoveryPassword(java.lang.String password)
  ```

- **setLogFile()**

  Unconditionally sets the log file name for this connection.

  ```
  setLogFile(java.lang.String logFile)
  ```

- **setNodelay()**

  Sets the `TCP_NODELAY` option for this connection. Toggling this flag can affect the performance of the application. If not set, it defaults to `true`.

  ```
  void setNodelay(boolean nd)
  ```

- **setPassword()**

  Sets the password for this connection.

  ```
  void setPassword(String p)
  ```

- **setPortNumber()**

  Sets the port number for this connection

  ```
  void setPortNumber(int pn)
  ```

- **setServerName()**

  Sets the server name for this connection.

  ```
  void setServerName(String sn)
  ```

- **setServicePrincipalName()**

  Sets the Service Principal Name for this connection.

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

- **setSSLConfigurationName()**

  Sets the SSL Configuration Name for this connection.

  ```
  setSSLConfigurationName(java.lang.String name)
  ```

- **setURL()**

  Sets the URL for this connection.

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

- **setUser()**

  Sets the username for this connection.

  ```
  void setUser(String u)
  ```